INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A MOON Simulator and Debugger

Chang Li

A Major Report
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

June 2002

ABSTRACT

A MOON Simulator and Debugger

Chang Li

The goal of this project is to design a simulator and debugger which extends the capabilities of the current simulator called MOON simulator into a new modern debugger with a friendly and nice looking graphical user interface. MOON is a programming language for a simplified RISC processor. It was designed as a target language for compilers written in the Compiler Design courses (COMP442 and COMP 642 in Concordia University), moreover, it can be used as an aid to learning assembly language concepts.

The MOON Debugger/Simulator can edit and assemble MOON programs into the "machine language" of the host processor, simulate the execution of programs on the processor, and provide some debugging facilities. The implementation is based on Java language with JDK1.3, and the system can run on any popular platforms.

This report covers the system requirement, GUI design, object-oriented design and implementation in Java. The user's manual and some class source files are listed.

ACKNOWLEDGEMENT

I am indebted to my supervisor, Professor Peter Grogono, for his guidance with patience and valuable time through this project.

I am grateful to my wife, Helen Hui Zhang, who supports me and gives me her best suggestions.

Table of Contents

Chapter 1: Introduction

1.1. Background

The MOON language is an assembly language designed for the MOON processor which is an imaginary processor based on RISC (Reduced Instruction Set Computer) architectures. It has a number of different simple instructions. All the instructions occupy one word and require one memory cycle to execute. The processor has a few instructions that access memory and many instructions that perform operations on the contents of registers. The memory address is a value in the range $0,1,\ldots,2^{31}$. Each address identifies one 8-bit byte. The processor can load and store bytes and words.

In 1995, Professor Peter Grogono wrote an assembler/simulator for MOON in C. MOON can be used for target language produced by the compilers written in the Compiler Design courses (COMP 442 and COMP 642) at Concordia University. This program can assemble a MOON program and simulate the execution of programs on the processor and provides rudimentary debugging facilities, however it is a command line program with a very primitive user-interface.

Recently, with the objected-oriented technology and graphical user interface design widely adopted for most software development, this MOON Debugger/Simulator needed to be rewritten to a new modern IDE (Integrated Development Environment) that should extend the capabilities of the existing assembler/simulator with a friendly GUI. Moreover, it should provide more debugging features and easy-to-use functions for users.

1.2. Goals

Based on the above rewriting demand for a new MOON Debugger/Simulator, a new IDE for MOON program should be developed. Since there are some requirements that the program runs on the platform independently, along with the application of Object-oriented technology and good GUI design, Java language is the one of the best choices. Java is a fully objected-oriented language with strong support for proper software engineering techniques.

Based on new Java technology, MOON can be re-engineered with several classes corresponding to the GUI frames, registers, memory, instructions, displaying tables, etc. Users can edit their MOON program code, compile code and run the compiled code in a simulated manner. In addition, users are allowed to modify values during the execution such as setting a breakpoint, and changing the content of the tables showing memory and registers would actually change the contents of the simulated memory and running results.

To achieve the goal, software engineering methods should be used in this software development. In the following chapters, a detailed software development process will be presented.

Chapter 2: Software Process Overview

Software development involves a set of activities and produces software as a result. The fundamental process activities include Software specification, Software implementation, Software validation and Software evolution. These activities can appear in different software development stages and they should be well defined. To achieve the goal to produce successful software, a good development process model is a must.

One of the first development process models offered is called the *waterfall model*. This model is linear, with one stage followed directly by the next. Each stage should be carefully completed before walking to next stages. We can extend this model by adding User-Interface Design process for this project with nice GUI. The complete model is depicted in Figure 2.1

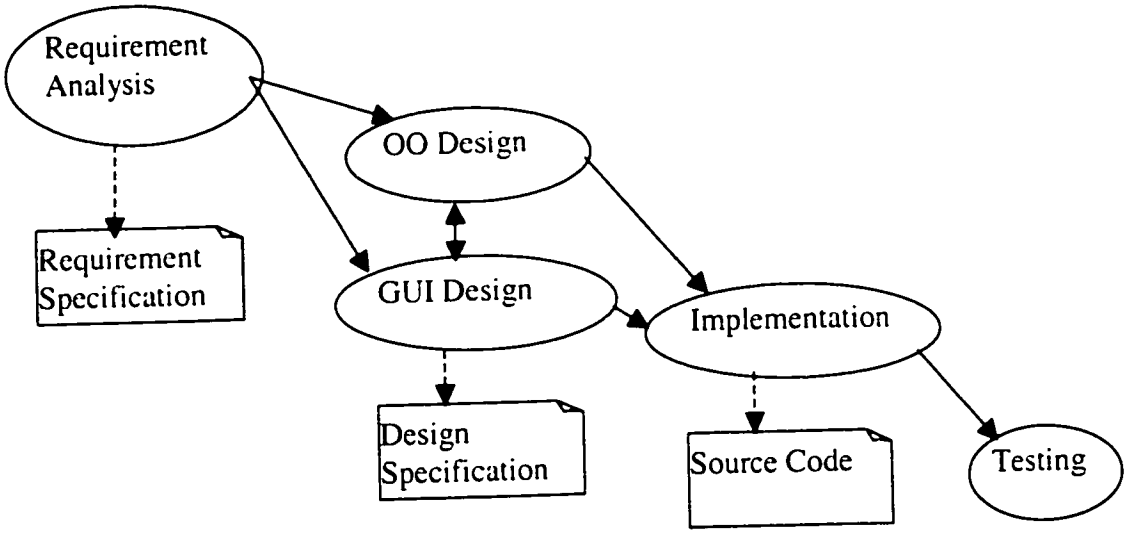


Figure 2.1

Requirement engineering process[3] is the first process. The MOON Debugger/Simulator has following principal stages in this process:

- Feasibility study (time, current software and hardware technologies)
- Requirements analysis (observation of existing system, discussions with users and experts, task analysis)
- Requirements definition and specification (gathering all the information in the analysis activity and writing the documents)

Software design starts from the requirements in natural language, via informal design to formal design. Since the late 1980s, Object-Oriented Design (OOD) has been widely publicized and adopted. The notations most relevant to OOD were recently standardized as the Unified Modeling Language (UML)[2]. UML can describe the classes and objects, visualize the behavior of a system, subsystem, or class.

Our system design also involves Graphical User Interface (GUI) Design. It includes task descriptions and usability goals, low and high fidelity prototypes, and the design evaluation.

Implementation is a concrete representation of the design. Successful design should produce nice implementation of classes and methods that are associated together.

Testing is the last step: it is used to find errors and measure the reliability of the program.

Chapter 3: Requirements Specification

3.1. Task Analysis

Task analysis is a process that builds a solid, extensible software design foundation. Often, software is designed by executing a series of prototypes. While prototyping is useful, the process does not provide any quantifiable way to ensure the design meets all the requirements or easily supports enhancements in the future. Task analysis provides a way to substantiate all aspects of the requirements by rigorously examining the user task flow. Task analysis also helps software designers get outside the box, by intentionally focusing on the operational problems to solve rather than implementation problems. The goal of task analysis is to empower the user beyond the original task requirements.

3.1.1. Task Goals

Based on analyzing the previous MOON Debugger/Simulator, the main task in the new system is to give the program a nice graphical user interface, more functions that user can use easily, and more debugging features. The new simulator can also aid in learning assembly language concepts. The program has been re-engineered in the OO style to improve the reusability and maintainability of the codes.

3.1.2. User Analysis

User Analysis is needed to know the people who will use the application and how they will use it. An approach to user analysis begins by observing users.

3.1.2.1. Users

- Students in the Compiler Design Course (Comp442 and Comp642)
- Other people who are interested in learning assembly language concepts.

3.1.2.2. Characteristics

Skills and Knowledge:

- Users should have enough skills and knowledge on the principles and practice of computer hardware and software.
- Users don't require special training on using this software.

Physical Attribution:

- Most users are aged between 18 and 50 and have university education level.
- Users have no serious health problem such as blindness or hand-disability.

3.1.1 Use Cases

To finally turn to implementation, it is required to drive the analysis of the system by identifying the system's use cases. A use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor. [2] This system use case is described in such a flow of events:

Moon Frame: The use case starts when the system shows a window with a menu bar called Moon Frame.

Source Editor: User can load and edit source code in this text editor.

Debugger: User can come into the debug frame and use some debugging facilities.

Parsing: User can compile the source code, and load the compiled code to the simulated memory.

Running: User can execute the instructions in the simulated memory in either of *Moon Frame* or *Debugger*.

Debug Facilities: From Debugger, user can trace instructions execution, set breakpoints, and change register and symbol table values.

The use case diagram is given in Figure 3.1.5.

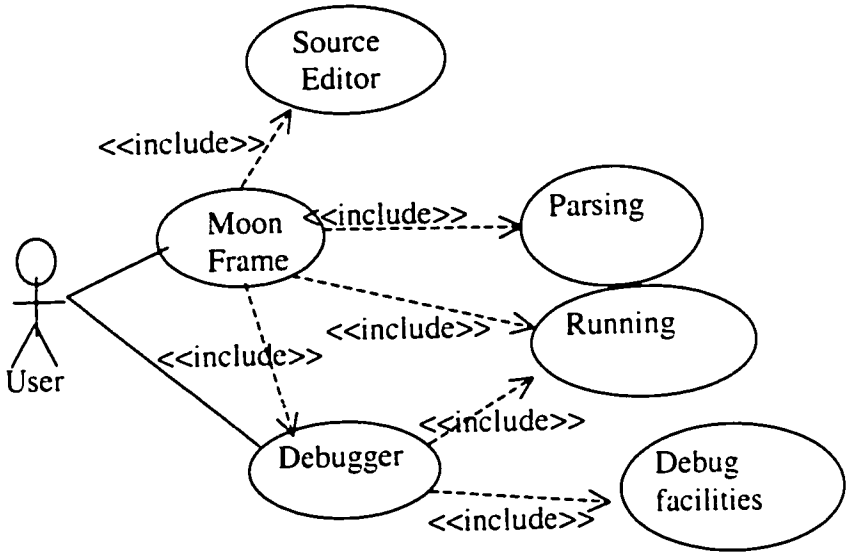


Figure 3.1.5

3.2. System functionality description

In this part, the system Functional Requirements will be presented in more detail. It has been mentioned above that the new MOON Debugger/Simulator extends the old system to a simple and modern debugger. The new features are added:

1. MOON program source code can be loaded and edited in the editor. In addition, more than one MOON program can be loaded and compiled all together.

2. The registers, memory, and symbols can be displayed on the same screen. In the debugger, users can access them to read or modify their values during execution.

3. Users can interact with the system: they can stop, pause and continue the execution or set up break points. Even if the user's MOON program has a non-breakable loop in the running without debugging, user still can make decision to stop the execution of the program.

3.3. Functional Specification

In this section, a more detailed description of each of the system functions will be given. These are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. [3] A template for describing the functions is introduced here. The template will have all of the following fields:

Function:	function's name
Description:	explanation of the function in detail
Input:	function's input
Output:	function's output

3.3.1. Main Frame

Function: Main Frame

Description: The system starts when it pops up the main frame with two windows: Editor and Output; menu bar and information bar. User can edit source code in Editor window. Output window is non-

editable. User can click one of menu items and their submenus to choose corresponding functions wanted.

Input: Text (MOON program source code) on the editor window

Output: Output stream on the output window and some information on the bottom text bar.

3.3.2. New

Function: New

Description: When the user clicks "New" submenu under 'File', all the text on the editor window will clear and it is ready for new program assembling and running. Before this action, if something has changed in the editor window, system will prompt "Save Changes?", user can choose "yes" or "no".

Input: None

Output: All text in the editor window is cleared.

3.3.3. Load

Function: Load

Description: When the user clicks "Load" submenu under 'File', it pops up a File chooser dialogue, and the user can choose a file in the file system, then the text of the file will appear on the editor window. If the user repeats this action, another file can be appended to the end of last file on the editor window. Before the "Load" action, if

something has changed in the editor window, system will prompt "Save Changes?" to the user who can choose "yes" or "no".

Input: File: path + name.

Output: Texts are appended to the editor window.

3.3.4. Save

Function: Save

Description: When the user clicks 'Save' submenu under 'File', the program saves the text contents in the editor window to the opened file.

Input: None

Output: Display message "Saved"

3.3.5. Save As

Function: Save As

Description: When the user clicks 'Save As' submenu under 'File', the program pops up a File chooser dialogue, and the user can choose a file or input a new file name in the file system, then the text in the editor window will be save to this selected file or new file.

Input: File: path + name.

Output: Display message "Saved as ..."

3.3.6. Exit

Function: Exit

Description: When the user clicks 'Exit' submenu under 'File', the system will terminate and close.

Input: None

Output: None

3.3.7. Cut

Function: Cut

Description: After the user highlights the text on the editor and clicks "Cut" under "Edit" menu, the selected text will be deleted from the editor and pasted to the clipboard.

Input: Highlights text

Output: None

3.3.8. Copy

Function: Copy

Description: After the user highlights the text on the editor, user clicks "Copy" under "Edit", the selected text will be cut to the clipboard.

Input: None

Output: None

3.3.9. Paste

Function: Paste

Description: After the user clicks "Paste" under "Edit", the text will be copied from the clipboard to where the cursor is on the editor window.

Input: None

Output: None

3.3.10. Select All

Function: Select All

Description: The user clicks "Select All" under "Edit", all the text on the editor window will be highlighted.

Input: None

Output: All the text on the editor window is highlighted

3.3.11. Assemble

Function: Assemble

Description: The user clicks "Assemble" under "Run", the text on the editor will be parsed. If some errors occur, the compiler will report the errors to the output window, and system will pop up a message dialogue to inform the user the assembling is not successful. If it is successful, the information bar will display "Assembly successful", and "Run" and "Debug" menu will become enabled.

Input: Text on the editor window

Output: Assembling result

3.3.12. Run

Function: Run

Description: The user clicks "Run" under "Run", the user's MOON program will run; "Stop" is enabled, other menus are disabled and the

output stream will go to output window while the input will be fetched from the prompting input dialogue in real-time. If there are some errors, the error message will be prompted in the message dialogue.

Input: Needed input by the running program

Output: Error message or running result.

3.3.13. Stop

Function: Stop

Description: The user clicks "Stop" under "Run", the executing program will stop immediately. Bottom information bar will display "Stopped"

Input: None

Output: Information bar display "Stopped"

3.3.14. Debug

Function: Debug

Description: The user clicks "Debug" under "Run", a debugger frame will appear on the screen.

Input: None

Output: None

3.3.15. Run

Function: Run

Description: The user clicks "Run" button, the compiled program will run and display result to the output window, while the register table will be refreshed according to the changing values of registers. If there are errors, the execution will stop and system will prompt an error message dialogue.

Input: None

Output: None

3.3.16. Looking for the memory with specified PC

Function: Looking for the memory with specified PC

Description: The user inputs the value of the Program Counter (PC) on the PC input box, and clicks "OK" button, then the corresponding memory content will display on the memory table and it will be highlighted.

Input: Program Counter (number)

Output: Memory word on the memory table

3.3.17. Set breakpoints

Function: Set breakpoints

Description: The user can toggle the check on the breakpoint column of the memory table to set/clear one or more breakpoints. User also can click a toggle button to set or clear all the breakpoints at once.

Input: Check/Uncheck on the check box or toggle button

Output: None

3.3.18. Change values of the registers or symbols

Function: Change values of the registers or symbols

Description: When the program is paused or before the execution of the program, the values of registers on the register table and the values of symbols (a symbol is similar to a "Label", representing the position of the corresponding instruction) on the symbol table can be changed by the user. The memory table will be changed in accordance with the above action when running. If input errors occur, an error message dialogue will be displayed.

Input: Register values and symbol values

Output: corresponding values in all the tables

3.3.19. Step running

Function: Step running

Description: Once the execution stops either by encountering a break point or by "Pause" button pressed, user can click "Step" button to make the execution continue. The user can achieve the execution in single steps by setting all the instructions with breakpoints.

Input: None

Output: None

3.3.20. Pause

Function: Pause

Description: When the program is running, the user can pause the execution at any time by clicking the "Pause" button, and the paused memory word will be highlighted at the memory table.

Input: None

Output: None

3.3.21. Stop

Function: Stop

Description: When the program is running, user can stop the execution by clicking "Stop" at any time, and the PC will jump to 0.

Input: None

Output: None

3.3.22. Help

Function: Help

Description: In the MOON main frame window, user can click "Help" on the menu bar to get help information.

Input: None

Output: Help contents display in a window

3.4. Non-Functional Requirement

This part define system properties and constrains on the services or functions offered by the system.

3.4.1. System software and hardware minimal requirement

- Unix, Linux, Windows95/98/2000/NT/XP etc. OS installed.
- 1.0M Hard disk.
- CPU 200M HZ
- 32M Memory
- JDK 1.2 or above installed

3.4.2. Ease of use

The system should provide good GUI so that user can use the system with satisfaction and explore the functions without any difficulties. Help menu is also necessary to be provided to help user understand the system and know how to use it.

3.4.3. Reliability

The system should run smoothly and reliably on most platforms. Any exceptions should be caught by the system and report to the user, then turn to the corresponding handler processes. GUI components should react to the events correctly.

Chapter 4: Graphical User Interface Design

Graphical user interface gives a program a distinctive "look" and "feel" and user can spend less time trying to remember which keystroke sequences do what and spend more time using the program in a productive manner.

MOON Debugger/Simulator GUI design follows the steps:

- Task analysis and usability goals
- Using the task descriptions to decide upon functional requirements.
- Low and high fidelity prototypes based upon the above.
- Evaluate the design through review and interviews.
- Consider architecture, design patterns and frameworks, UI Toolkit, etc.

In the requirement engineering process, we have finished task analysis, and requirement definition and specification. They can be mapped to the first two steps of GUI design, so we only present the last three steps here.

4.1. Prototypes

Prototypes are the process of building a concrete and visible design solution starts from user requirement descriptions. Prototypes can be used in this process to validate/review requirements with end-users, to gather further information about the system, to estimate/validate the design difficulties or decision, to investigate different design solutions[7]

Using JBuilder4.0 as a rapid prototyping tool enables a high fidelity prototype to be obtained.

4.1.1. Main Window

When user starts the program, it appears the following frame, which includes two text windows, one menu bar and a message bar at the bottom.

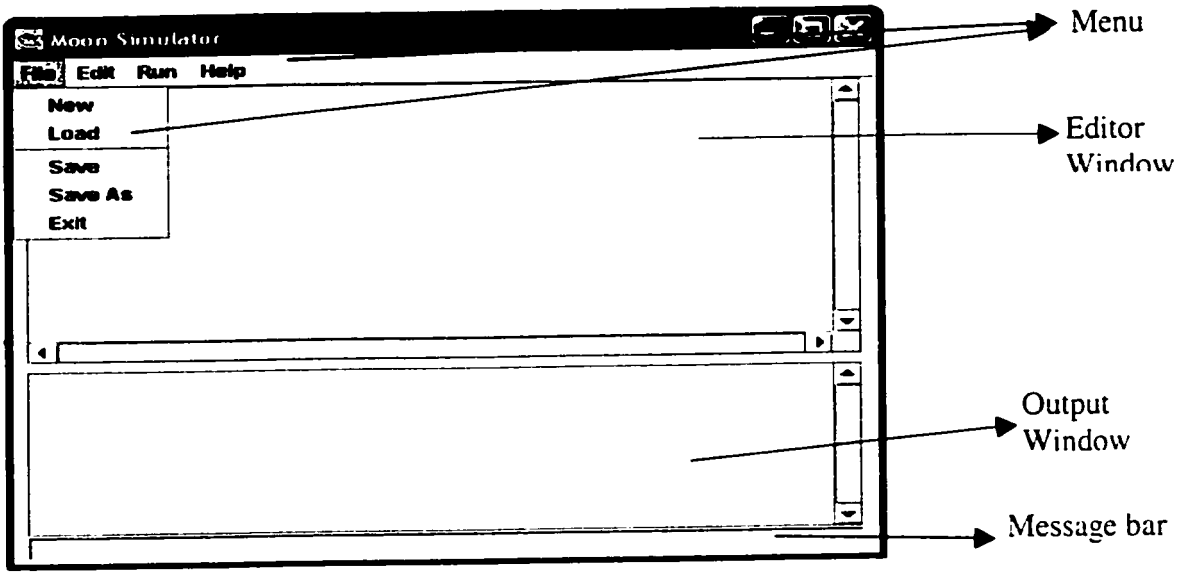


Figure 4.1.1

4.1.2. File Menu

If the user clicks "Load" below "File", a file chooser dialogue will be displayed.

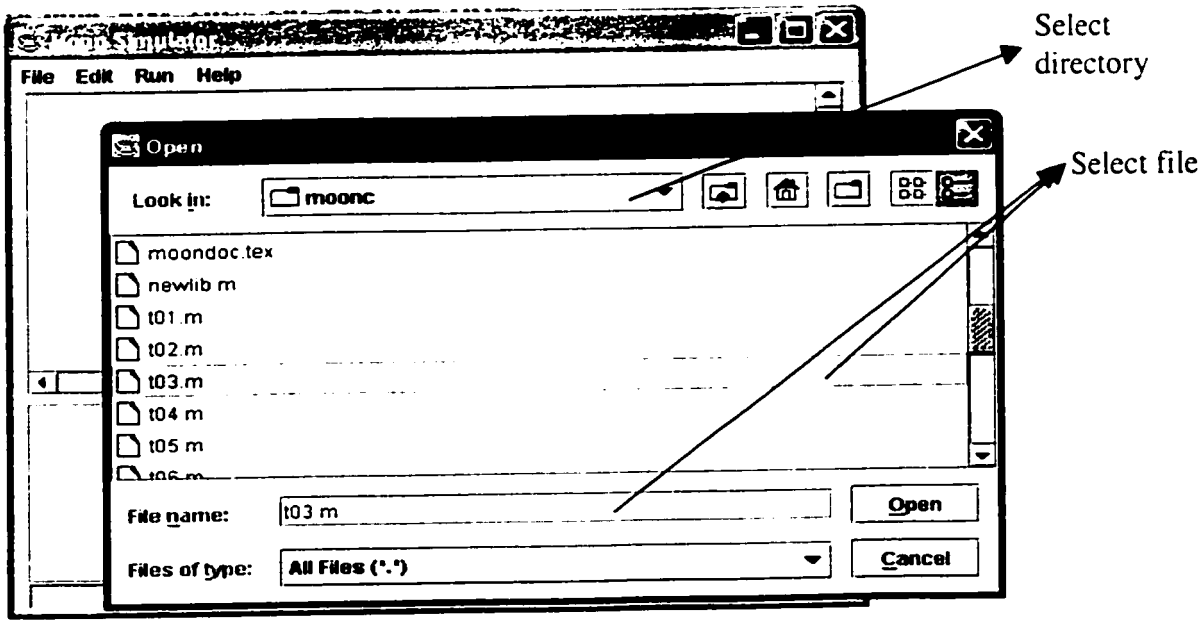


Figure 4.1.2

4.1.3. Edit Menu

The user can edit the text on the editor window.

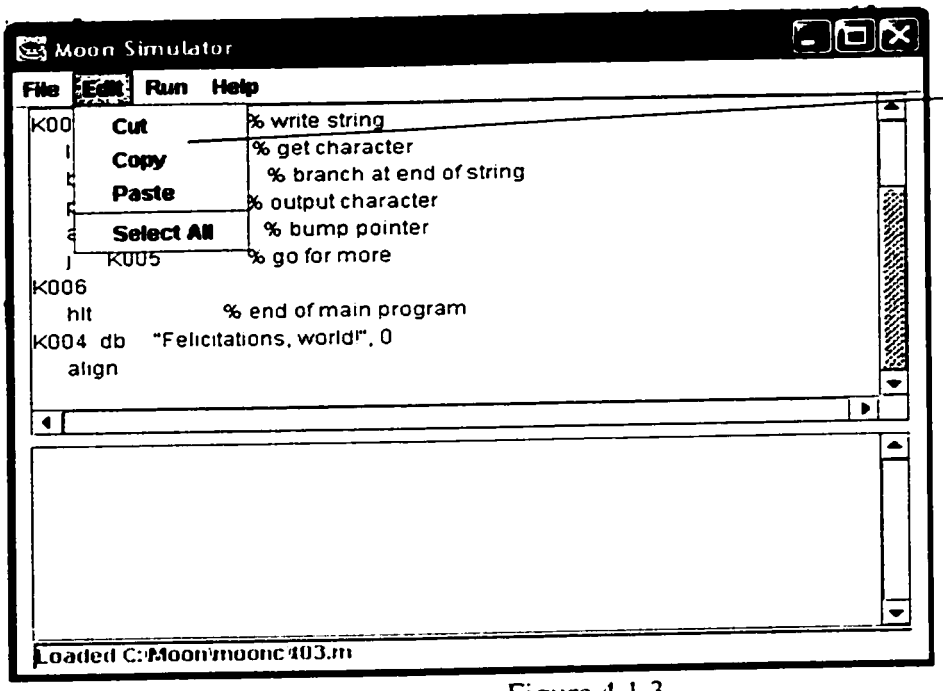


Figure 4.1.3

4.1.4. Run Menu

The user clicks "Assemble" under "Run". If it has error, system shows this screen.

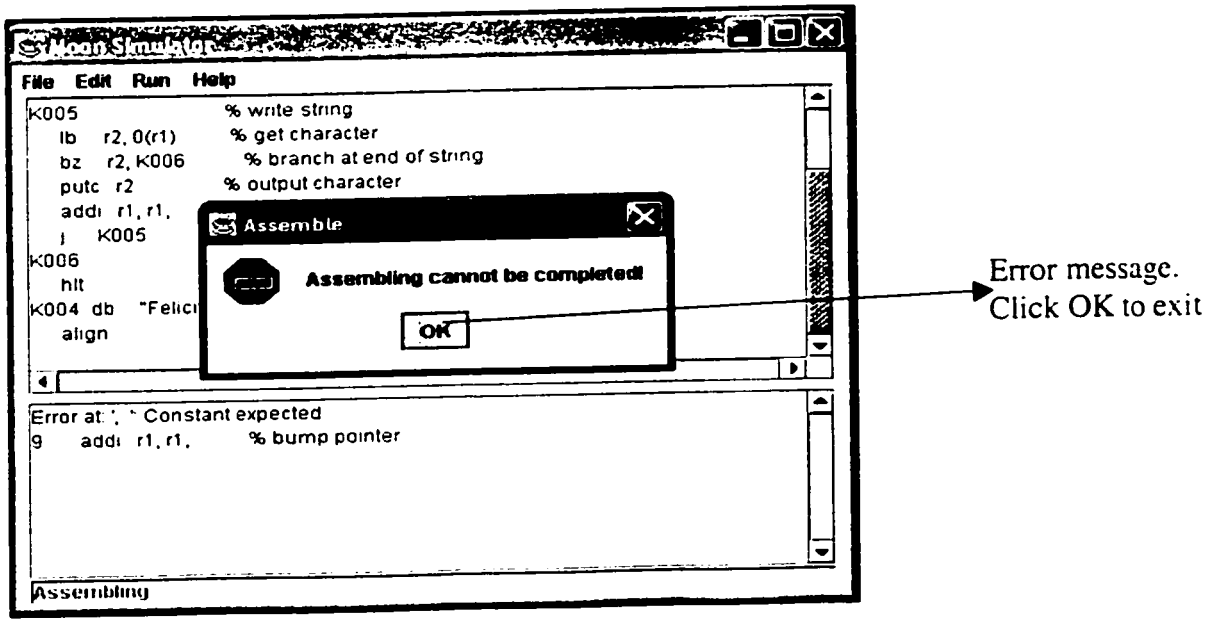


Figure 4.1.4

4.1.5. Assemble Error Dialogue

If user wants to run "Debug" before code has been modified and not been compiled, a prompt dialogue will be given to inform that case.

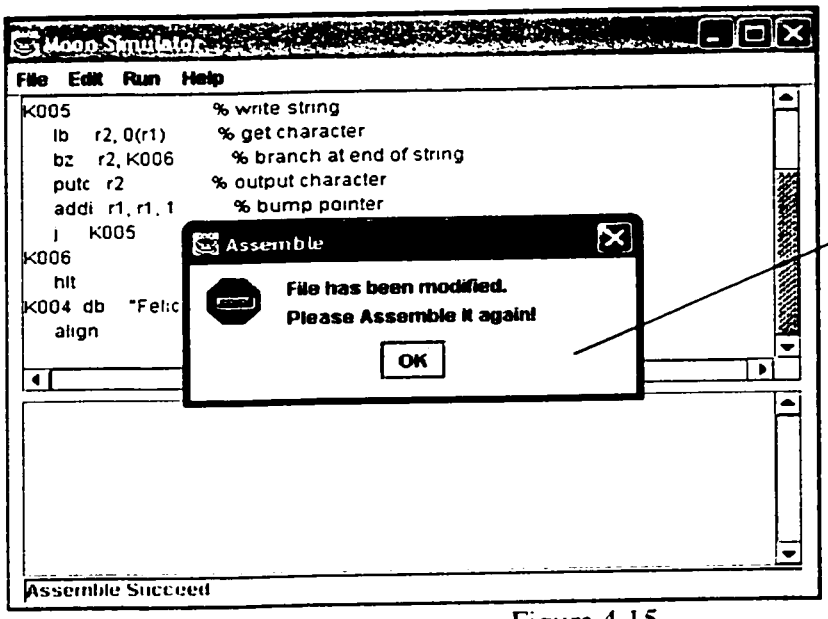


Figure 4.15

4.1.6. Debug Window

If code has been compiled successfully, and the user clicks "Debug" under "Run", a new frame appears.

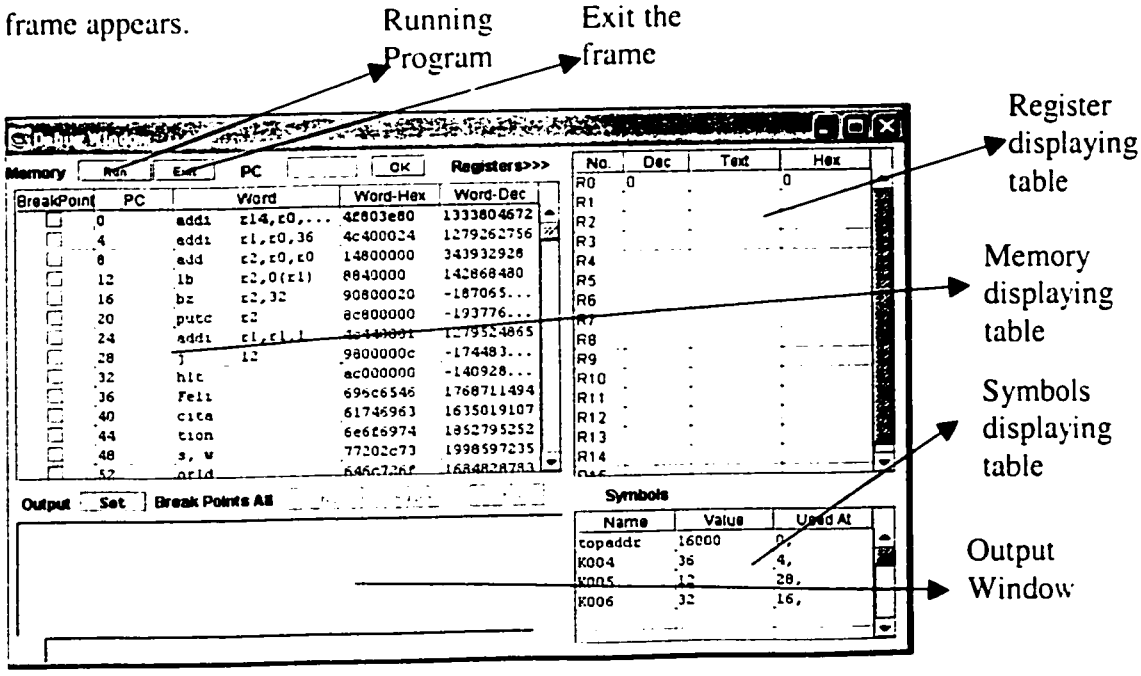


Figure 4.1.6

4.1.7. Run Button

The user clicks "Run" button, the code will run: the register's values will display on the register table; output result will be on the output window;

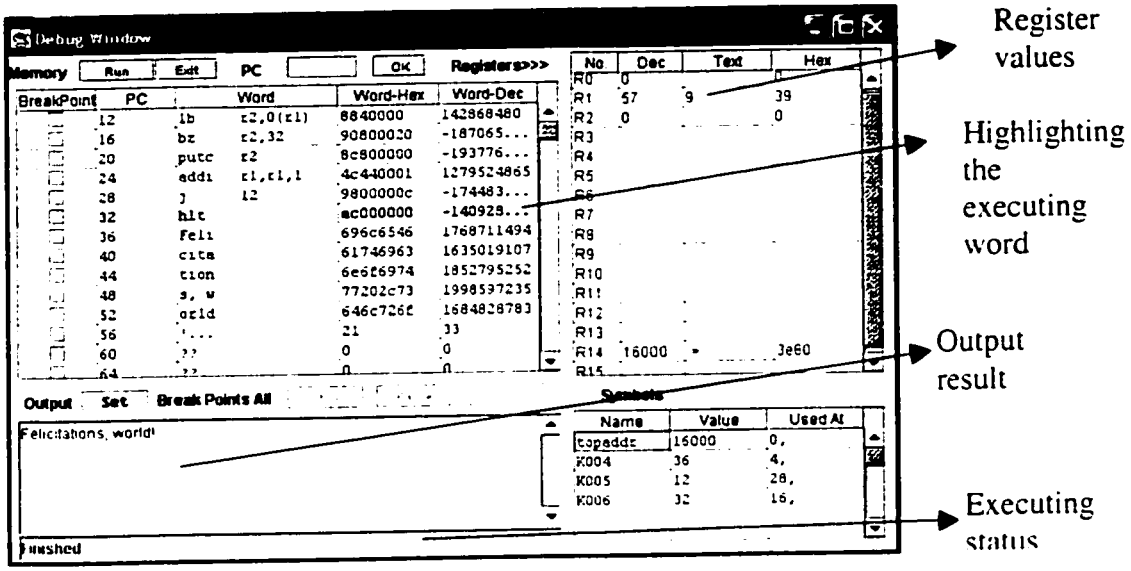


Figure 4.1.7

4.1.8. Other Buttons in Debug Window

The user can set/clear one or more breakpoints in the memory table by clicking the check boxes or set/clear breakpoints of all the instruction words.

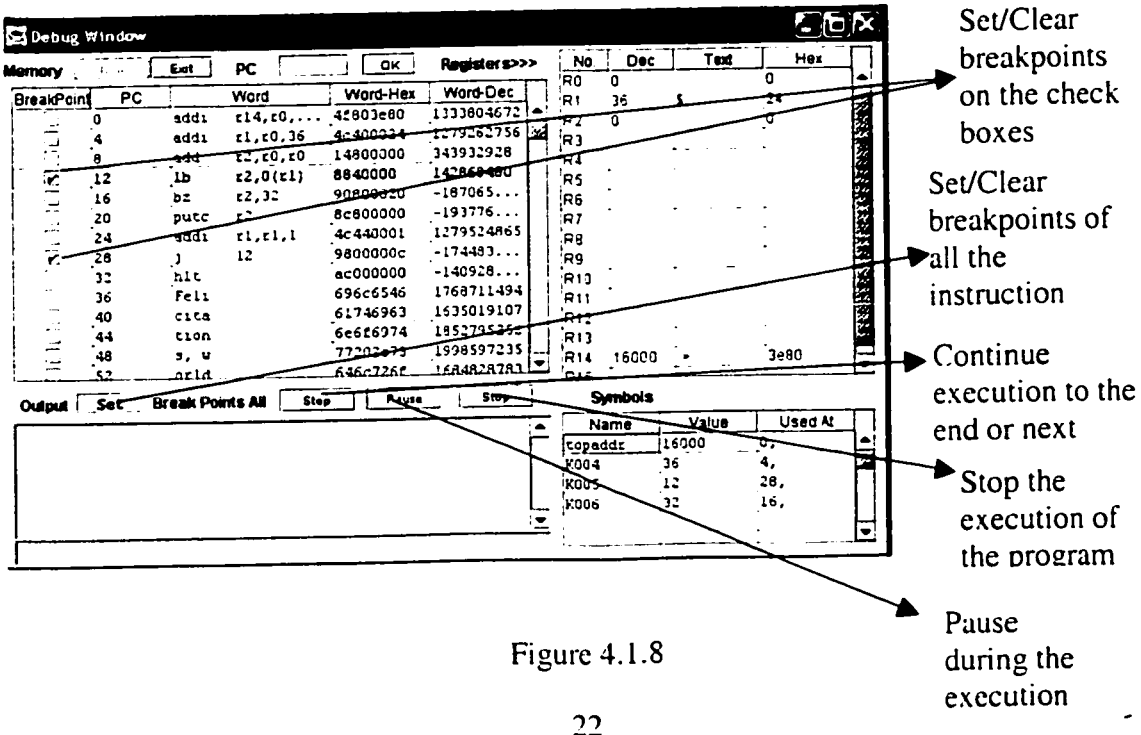
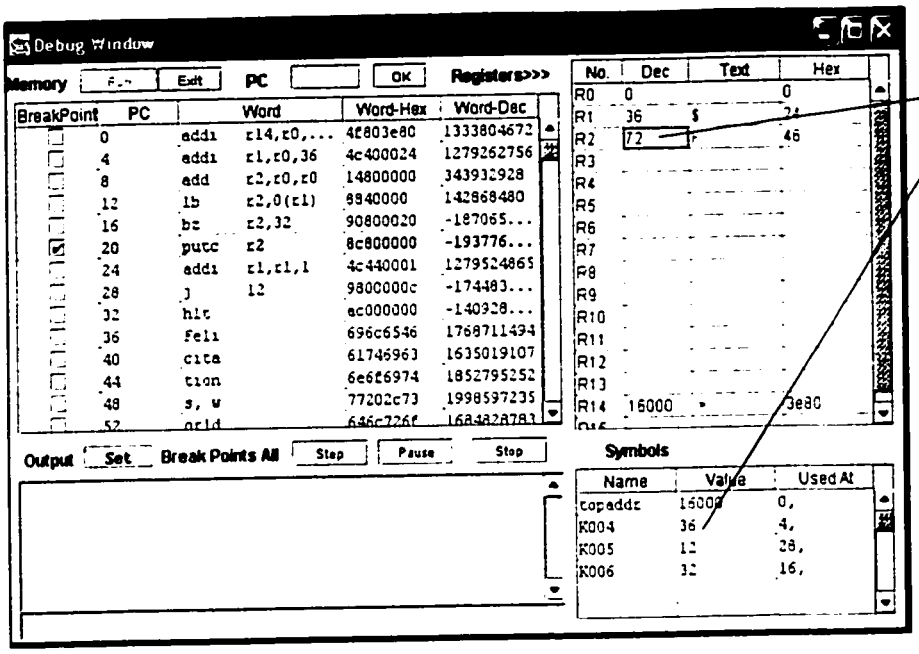


Figure 4.1.8

4.1.9. User Updates Tables

Before the execution, at breakpoint or during the pause, user can modify symbols and registers' value.

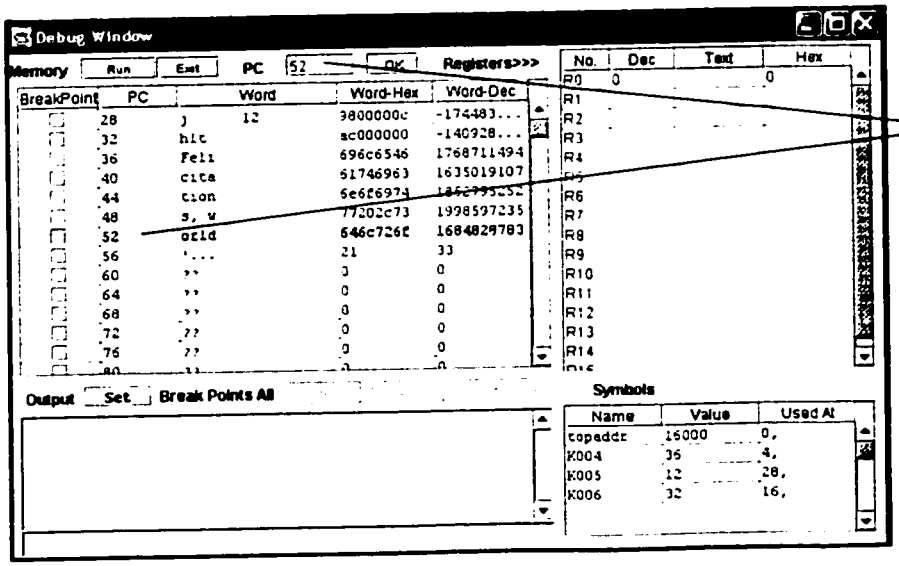


User update the value of register, then press "Enter" to apply.

Figure 4.1.9

4.1.10. Program Counter

The user can watch any memory word within the memory table.

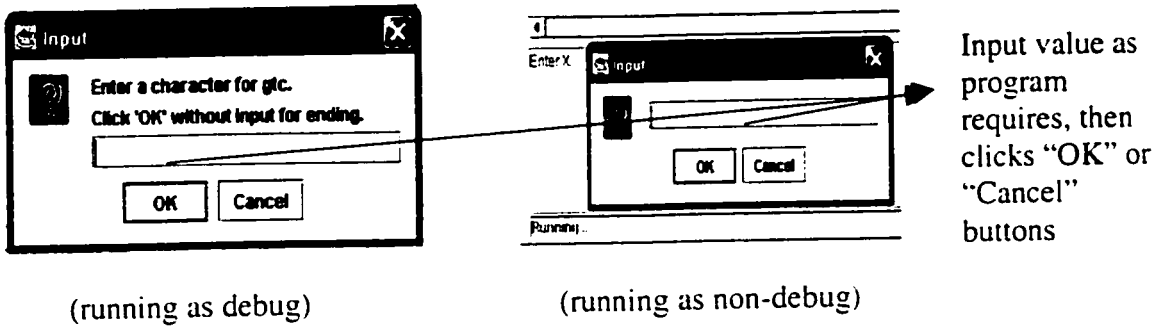


Input number (Program Counter), then press "OK" button, the window will display the corresponding memory content.

Figure 4.1.10

4.1.11. Input Dialogue

If the program execution needs the user to input some values, the input message dialogue will appear as follows:



(running as debug) (running as non-debug)

Figure 4.1.11

4.1.12. Error Message Dialogue

If the user inputs wrong format values in editable table, an exceptional message dialogue will prompt.

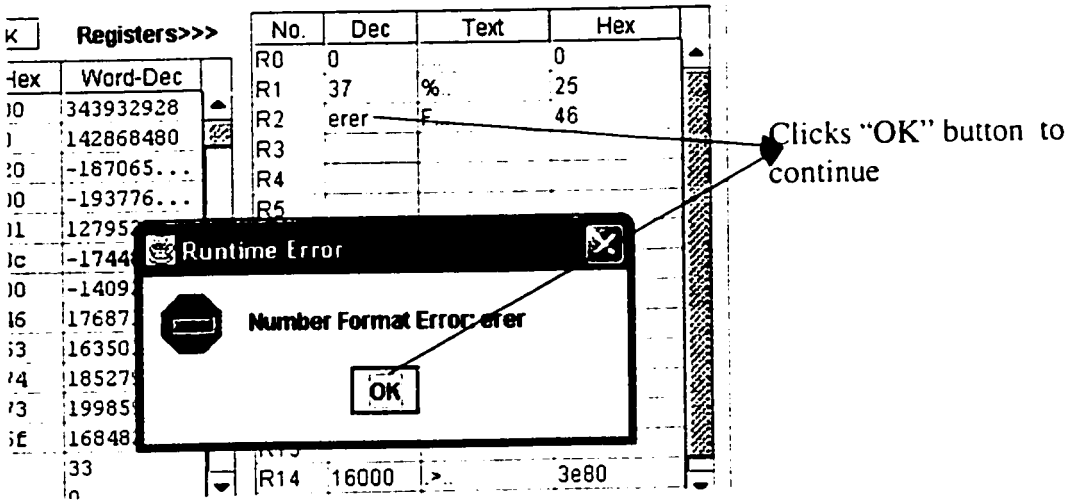


Figure 4.1.12

4.2. Evaluation of Prototypes

After the completion of the prototype, it should undergo usability inspection by usability specialists (software developers, users and other professionals), to identify any problems and check for conformance with style guides as well as established ergonomic principles[7].

This project follows heuristic evaluation, developed by Jacob Nielsen and Rolf Molich, which is a guideline or general principle or rule that can be used to guide a design decision or to criticize a design that has already been made.[9]

According to the survey on current Java IDEs on the report by A.Seffah and J.Rilling [8], the top five problems of usability of the GUI identified are:

- Making visible program artifacts (classes, interfaces, structure, etc.) and IDE functionalities when they are relevant and required.
- Minimizing the developer's memory load.
- Speaking the developer's language.
- Keeping the developer informed on the IDE status and the program being constructed.
- Preventing developers from making mistakes.

These problems can be regarded as our interface design evaluation principles. Five master students in computer science department of Concordia University as human factors engineers examined the interfaces. Based on their aggregated findings, the potential problems of interfaces were discovered and the design was iterated.

4.3. MVC and Java GUI components

The Java Foundation Classes (JFC) software extends the original Abstract Window Toolkit (AWT) by adding a comprehensive set of graphical user interface class libraries. Swing is a graphical user interface (GUI) component kit, part of the Java Foundation Classes (JFC) integrated into Java 2 platform, Standard Edition (J2SE). Swing simplifies deployment of applications by providing a complete set of user-interface elements written entirely in the Java programming language. Swing components permit a customizable look and feel without relying on any specific windowing system. Swing is incorporated in the Java 2 platform, there is no need to download or install it. [10]

Because of the mentioned characteristics of Swing, this system interface implementation uses Java Swing as the main tool. Next, let's take look at the important concepts behind Swing, and how they affect this system development.

4.3.1. MVC

MVC (the Model-View-Controller) is the underlying architecture of Swing. It defines how the classes that make up a user interface should be structured and how it achieves pluggable look and feel.

MVC separates a user interface into three classes. The model class encapsulates the data underlying the interface, and has no knowledge of how the data will be displayed, or how the UI will modify it.

The view class displays the data contained by a model. In the context of Swing, the separate view class allows us to change the look of a component without changing its underlying data model.

The controller class manages the interaction with the user. It modifies the data model in order to update its contents, and refers to the view in order provide feedback to the user.

4.3.2. Swing Components

Swing components are written in the Java programming language, without window-system-specific code. This facilitates a customizable look and feel without relying on the native windowing system, and simplifies the deployment of applications.

Swing components are often referred to as *lightweight* components. *Lightweight* component is one that "borrows" the screen resource of an ancestor (which means it has no native resource of its own -- so it's "lighter"). Unlikely, AWT components are called *heavyweight* components. A *heavyweight* component is one that is associated with its own native screen resource (commonly known as a *peer*). The classes of Swing components are from package **javax.swing**. In applications with **JFrame**s, we attach components to **Container** which is a collection of related components. **JComponent** is a subclass of **Container** rather than Component, which gives it the ability to add child components.

These components are used in MOON system:

Buttons: JButton, JToggleButton

Menus: JMenuBar, JMenuItem

Tables: JTable, DefaultTableModel, AbstractTableModel

Text Components: JTextArea, JTextField, Document

Common Dialogs: FileChooser

Bar: JScrollBar

Lables: JLable

4.3.3. Swing Component Construction

The Swing component is created by passing in a data model object to its constructor, then the component requests a delegate from the **UIManager** class. The **UIManager** returns a delegate that is suitable for the current look and feel[6]. Following is the code for the "look and feel" as **Metal** in MOON system:

```
try {
      UIManager.setLookAndFeel(
       "javax.swing.plaf.metal.MetalLookAndFeel");
    }
    catch(Exception e) {
      e.printStackTrace();
    }
```

Figure 4.3.3 shows the construction of a Swing component:

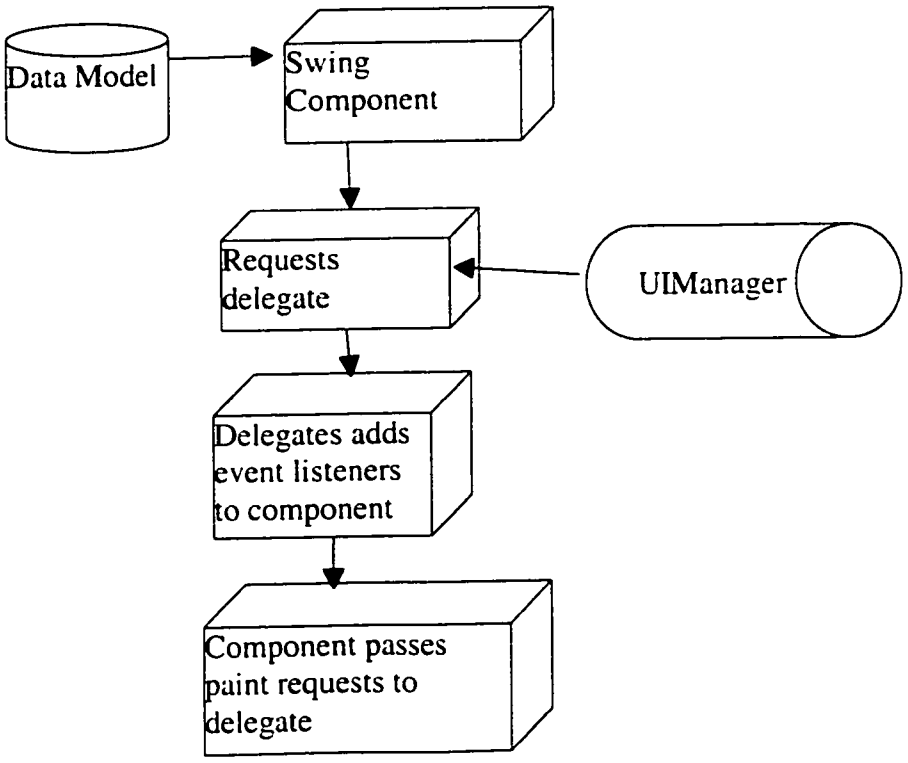


Figure 4.3.3

Understanding MVC and Java GUI components can help us to do better work in the following software development processes. With Java AWT, Swing *lightweight* components and the underlying structure-MVC, the object-oriented design and the implementation of the system will become easier.

Chapter 5: Object-Oriented Design and Implementation

Object-Oriented Design (OOD) centers on finding an appropriate set of classes and defining their contents and behavior. It involves determining the proper set of classes and then filling in the details of their implementation. Object-oriented design is fundamentally a three-step process: identifying the classes, characterizing them, and then defining the associated actions.[5]

MOON Debugger/Simulator is developed with OOD and implemented in Java. It is composed of variety classes and can be inherited and extended.

5.1. Understanding The Problem

The first step to make object-oriented design is to understand the problem. This system is about an assembler language simulator and debugger, hence the main problems are concluded as:

1. How to edit the source code
2. How to compile the code
3. How to run the program
4. How to debug the compiled code

Some problems are complex and they will probably be divided to more sub problems; such as "How to debug the compiled code" can be grouped by "How to set and clear breakpoint", "How to change the register value", "How to display the memory", etc.

On these problems, some criteria for correctness should be established that will tell us if the system is working:

1. Source code can be displayed, edited and saved.
2. Before running, the code must be compiled successfully.
3. If execution has error, it should be stopped.
4. Because the simulated memory is 4000 words, MOON program source code should be not more than 4000 lines in this system.

5.2. Identifying Candidate Classes

Based on the problems understood and some sense of how they should be solved, a set of classes can be identified.

Class with main function:	Program can be executed from running this class in OS.
MOON Frame:	Interface for source code editing and compiling, and MOON Program simulated execution.
Debugger Frame:	Interface for code debugging facilities.
Source Editor:	A text Editor for MOON program source code.
Source Parser:	A compiler for assembling the source code to the simulated memory and checking non-real-time errors.
Program Executor:	It can simulate the program execution and run in either of normal way or debugging way
Memory:	Simulated MOON memory content with access methods
Register:	Simulated 15 registers of MOON with set and get methods

Symbols:	Set symbols value and addresses where they occur
Instruction:	MOON instruction set and instruction access.
Word Types:	Memory word types (Format A, Format B and data) and their access.
Displaying Tables:	Displaying table such as registers, symbols and memory table data models construction
Token and Token Types:	Kinds of tokens for source code parsing.
Operation Code Types:	Operation codes enumeration

5.3. Class Architectures and Implementations

MOON Debugger/Simulator is an IDE which includes user interface, editing, parsing, executing and debugging etc. components. Some of these components can be grouped within a class and some can be dependent classes. The relationships between these classes are complicated and each class has its own attributes and operations. Here, we use Unified Modeling Language(UML) to describe our classes and their relationships. UML is a standard language for writing software blueprints and it may be used to visualize, specify, construct, and document the artifacts of a software-intensive system[2].

5.3.1. MoonClass

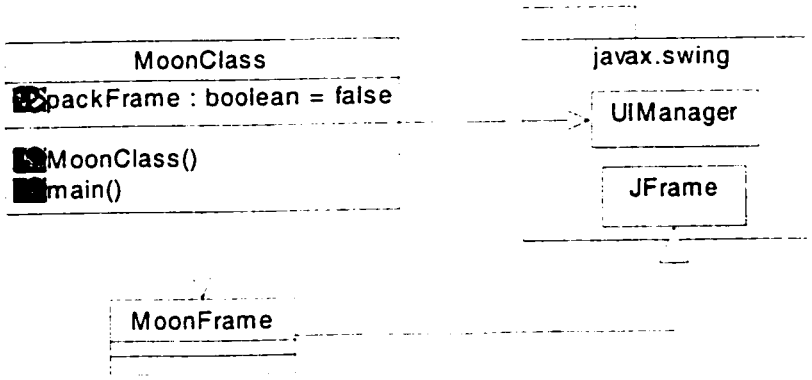


Figure 5.3.1

Figure 5.3.1 shows the class structural diagrams of **MoonClass** and **MoonFrame**.

MoonClass is the "first" class that involves the static main operation "public static void main(String[] args){...}" and instantiated an object of **MoonFrame** which provides the interface and other utilities. When user execute this file using "java moon.MoonClass" command line with JDK1.2 above installed, the program will be running.

5.3.2. Memory

Memory class(Figure 5.3.2) is constructed for simulating the main memory of MOON architectures. A memory address is a value in the range 0,1,..., 2^{31} . Each address identifies one 8-bit byes. The addresses 0,4,...,4N are word addresses. MOON Debugger/Simulator can simulate memory size up to 4000 words. The memory can not be enlarged over 4000 words, because the memory display table is implemented in Java which gives much capacity limitation.

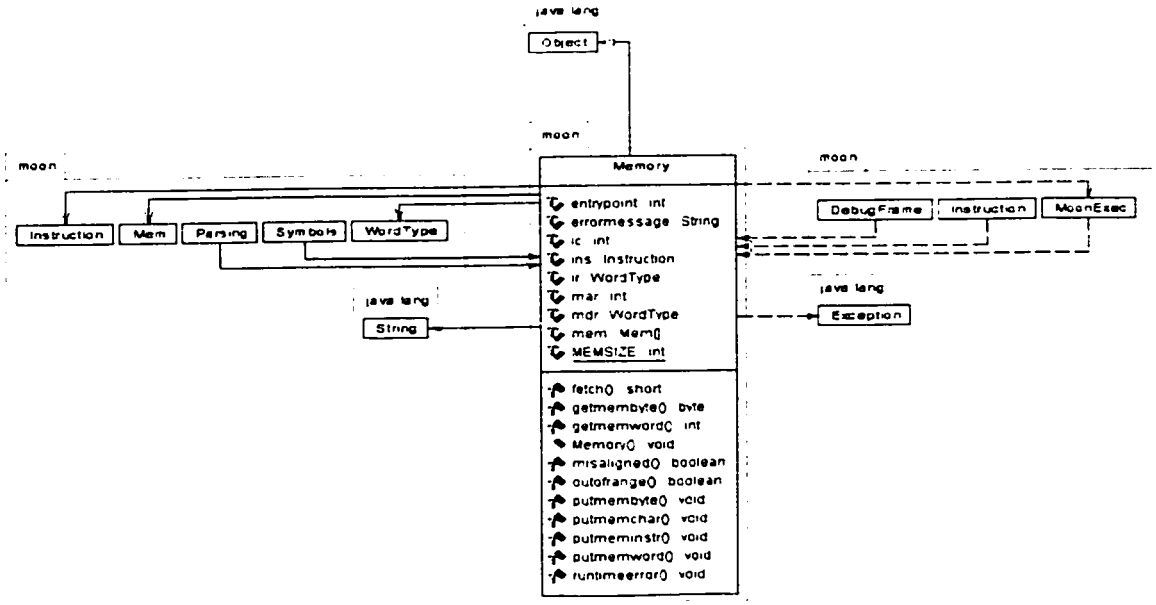


Figure 5.3.2

5.3.2.1. Instruction formats

Each instruction occupies one 32-bit word of memory. There are two instruction formats, A and B, shown in Figure 5.3.2.1. Both formats contain a 6-bit operation code. Format A contains three register operands and Format B contains two register operands and a 16-bit data operand. If an instruction contains a ***K*** operand, it can be seen as Format B.

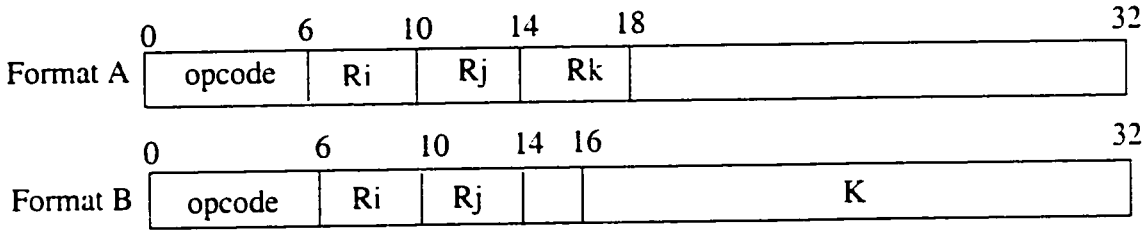


Figure 5.3.2.1

5.3.2.2. WordType

A word of simulated memory has four types: format A of instruction, format B of instruction, four bytes data and 32-bit signed integer.

The class (Figure 5.3.2.2) is created according to these four types of memory word. It has an integer attribute data (in Java, integer has 32-bit length) that is the unique simulated memory word area and it can be accessed from a series of methods *get* and *set*. With java bit shifting operators: the left shift operator(<<), the right shift operator with sign extension(>>) and the right shift operator with zero extension(>>>), the series of methods *get* and *set* can be achieved. Take an example, to set a 4-bit register operand in format A instruction can use method *set_fmta_ri(int bits)* :

```
public void set_fmta_ri(int bits) {
      Word = (bits<< 22) | (0xFC3FFFFF & Word);
//register i: 4 bits
  }
```

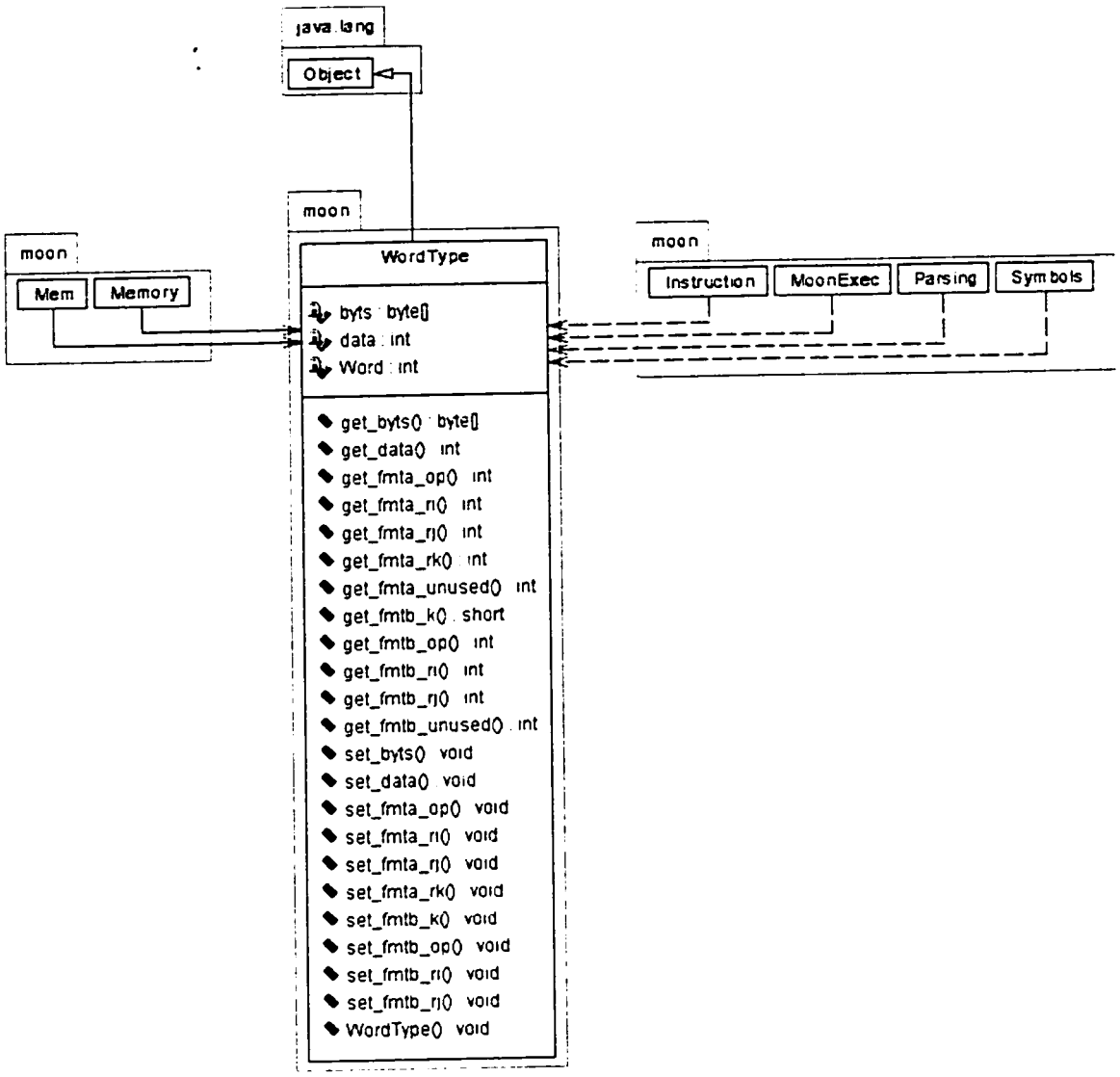


Figure 5.3.2.2

5.3.2.3. Memory Access

The simulated memory array of class **Mem** contains two flags, to tell whether the word contains an instruction and whether it is a breakpoint. Simulated addresses are byte addresses. Consequently, all addresses are shifted right 2 bits before being used. Only the memory module should know about this.

In **Memory**, there are eight methods for accessing, one constructor and two methods for handling exceptions.

- *Constructor:*
  1. Initialize Memory content by creating an array of **Mem** with size 4000.
  2. Create an instruction register(*ir*) as an instance of **WordType** which contains the instruction that will be executed next.
  3. Create memory data register(*mdr*) as an instance of **WordType**.
  4. Create memory address register(*mar*) to store word address of last access
  5. Create instruction counter(*ic*) which contains address of instruction that will be executed next.
  6. Instantiate an object of **Instruction.**
  7. Set *entrypoint* to –1 as the first instruction
  8. Initialize the error message string to blank.
- *outofrange()* and *misaligned()*:
  1. If address is illegal, *outofrange* will output error message to handler.
  2. If a memory word access is not on a four-byte boundary, *misaligned* reports error to handler.
- *fetch()* is to fetch the instruction at *ic*, store it in *ir*, and increment *ic*.
- *getmemword()* and *putmemword()* are to fetch a data word from memory and return it, and store a word in memory.
- *getmembyte()* and *putmembyte()* are to fetch a byte from memory and store a byte to memory
- *putmeminstr()* is to store an instruction to memory.
- *putmemchar()* is to store a character in memory.

5.3.3. Instruction

As mentioned in 5.3.2.1, there are two instruction formats, format A and format B. Instructions are divided into three classes: data access, arithmetic, and control.

We create class **OpType** that make all the operands types enumerated:

```
public class OpType {
     public static final int bad  = 0;
     public static final int lw   = 1;

     ...
}
```

In **Instruction** class, we define a string array to contain all the operand type string names that are corresponding to the enumeration of **OpType** by the array subscripts.

```
final static String opnames[] ={
  "","lw","lb","sw","sb","add","sub","mul","div","mod",
  "and","or","not","ceq","cne","clt","cle","cgt","cge",
  "addi","subi","muli","divi","modi","andi","ori","ceqi",
  "cnei","clti","clei","cgti","cgei","sl","sr","getc",
  "putc","bz","bnz","j","jr","jl","jlr","nop","hlt",
  "entry","align","org","dw","db","res"
};
```

Figure 5.3.3 shows class **Instruction**. This class is mainly functioned to display two types of instruction (Format A and Format B) or data. The method *showword()* including *showfmta()* and *showfmtb()* can decide which kind of instruction or word data is shown.

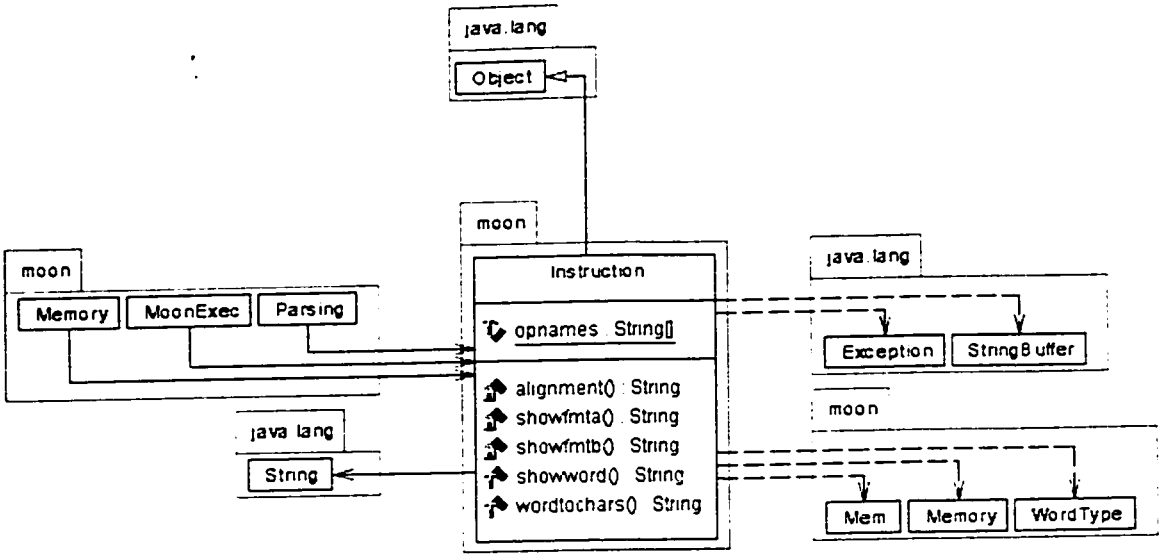


Figure 5.3.3

5.3.4. Symbols

A *Symbol* is a string consisting of characters: letters, digits, and _. The first character of a symbol must not be a digit. Directives and instruction codes must not be used as symbols. The predefined symbol *topaddr* is the highest legal address + 1 = 16000.

Class **Symbols** (Figure 5.3.4) is used for construction of a list of symbols and provide functions for storing symbols, getting symbols, finding a symbol and validating symbols.

The element of **Symbols** list is **Symnode** which records the name and value of a symbol and how often it has been defined (once is correct). The **Symnode** also contains a list of *uses* (instance of **LinkedList** which is from **java.util** package of the Java API for implementing and manipulating linked lists that grow and shrink during program execution), each of which contains an address where the value of the symbol is used.

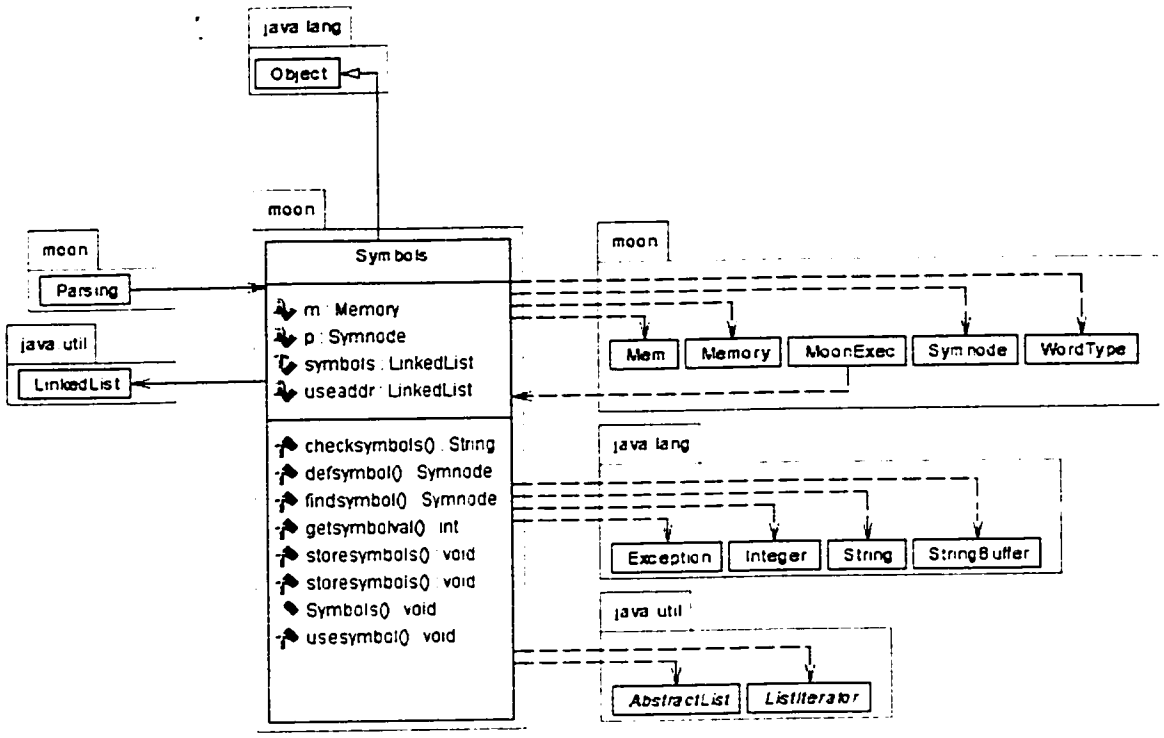


Figure 5.3.4

The constructor of **Symbols** is to define *topaddr* with method *defsymbol(String name, int Value).*

- *checksymbols()* is to check if there is something wrong about symbols.
- *defsymbols()* is to update a symbol that contains name and value. If it can be found with name by *findsymbol()* in the symbols list, the value will be updated. On contrast, the new symbol can be created if its name is not on the list.
- *usesymbol(String name, int addr)* is to add the *name* to the *uses* list at the *addr* that it uses.
- *getsymbolval()* is to return the value of a symbol, or -1 if it doesn't exist by iteration the link list of symbols.
- *storesymbols(String name, int val)* and *storesymbols()* are overloaded. They can store symbols to the list.

5.3.5. Parser

The class for parsing the source code and assembling it to the memory is **Parsing** (Figure 5.3.5).

5.3.5.1. Token and TokenType

Class **Token** contains information of token type, token value, register number, operation code and its position. **TokenType** defines in total nine types: T_BAD, T_REG ,T_OP, T_SYM, T_NUM, T_STR, T_COMMA, T_LP, T_RP, T_NULL.

5.3.5.2. Initialization

Class **Parsing** creates instances of **Memory**, **Token**, **Symbol** and **Instruction**. It also makes declaration of two variables as memory table model and symbol table model.

5.3.5.3. Constructor

The text from source code editing window is transferred to the constructor of **Parsing** as an argument, then it is tokenized by *StringTokenizer()* that will use delimiter string"\n\r" consisting of a newline and a return. It invokes *readline()* to parse string line by line. Next, the symbols are stored in the linkedlist with the method *storesymbols()* of **Symbols**. Finally, the parsed MOON program will be loaded to the simulated memory table and symbols table by *dump()*.

5.3.5.4. getint (), getop(), getreg(),getshort()

- *getint()* is to make the integer value of register token or symbol token. Then it invokes the *next()* to continue the parsing.
- *getshort()* is simlar to getint(), but it checks that the value only is stored in 16-bits.
- *getop()* is to parse operand code and return it.
- *getreg()* is to parse a register and return the register number.

5.3.5.5. isreg() and issymchar()

- *isreg(String name)* is to judge if the *name* is a valid register.
- *issymchar(char c)* is to return true if the *c* occurs in the symbol.

5.3.5.6. dump()

1. Create the instances of **MemDataModel** and **SymDataModel** representing memory table and symbols table.
2. Create a instance of **Vector** as a row of memory table, and insert data to the row.
3. Create an iterator of symbols linkedlist, then traverse it and insert data to the row of symbols table.

5.3.5.7. match()

- match the token , if something wrong, report a syntax error.

5.3.5.8. next()

1. Get a token starting from the end of last token, removing white space(' ' or '\t') between tokens.
2. If the first character of the token is a letter, read the token as register, symbol or operand code and return it. If not, continue.
3. If the character is a digit or sign "+" or "-", read following digits as decimal integer value. If not, continue.
4. If the character is ' " ', read a character string enclosed in quotes. If not, continue.
5. Continue check if the character is ' ,', ')', '(', ' ' or '%' and get its type of token and value of token.
6. Continue to 1 until the end of one line.

5.3.5.9. readline()

1. If a token is a symbol, the symbol will be defined in the symbols linkedlist.
2. If a token is an operand code, the corresponding operation will be given.
3. If there is an exceptional condition, throw the exception to handler.

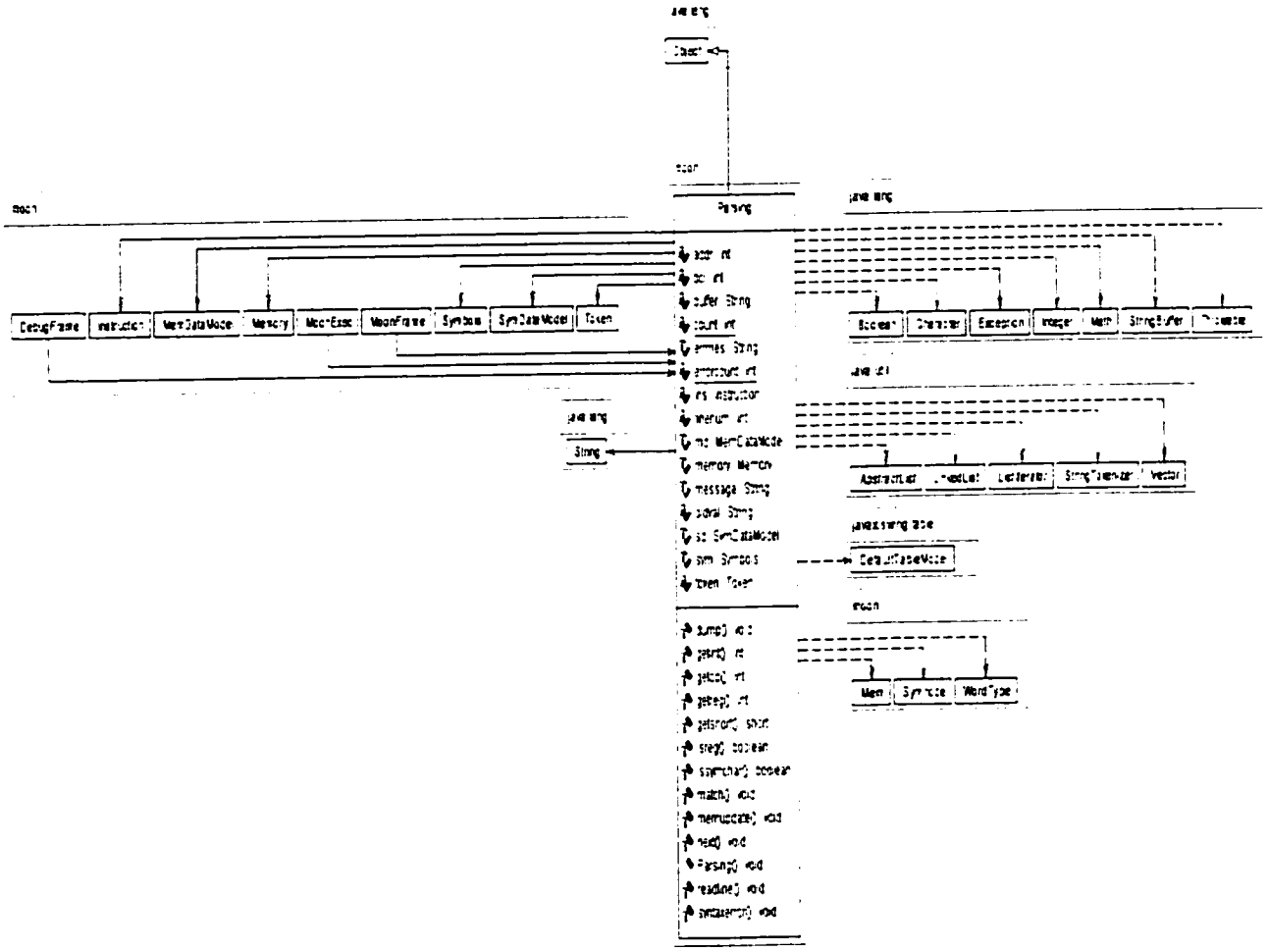


Figure 5.3.5

5.3.6. Registers

MOON has 16 registers denoted R0, R1, ..., R15. Each register can store a 32-bit word. At all times, R0 is always 0. **Registers**(Figure 5.3.6) is the class for simulating the registers of MOON. It contains an array of Integer with size of MAXREG=16 to

represent total 16 registers and three methods. Two of them are *fetchreg(short regnum)* to get the value of a register, and *storereg(short regnum)* to set the value to a register. The other method is *runtimeerror(String message)*, and if some runtime error occur during the execute the previous two methods, it will report the error to the handler.

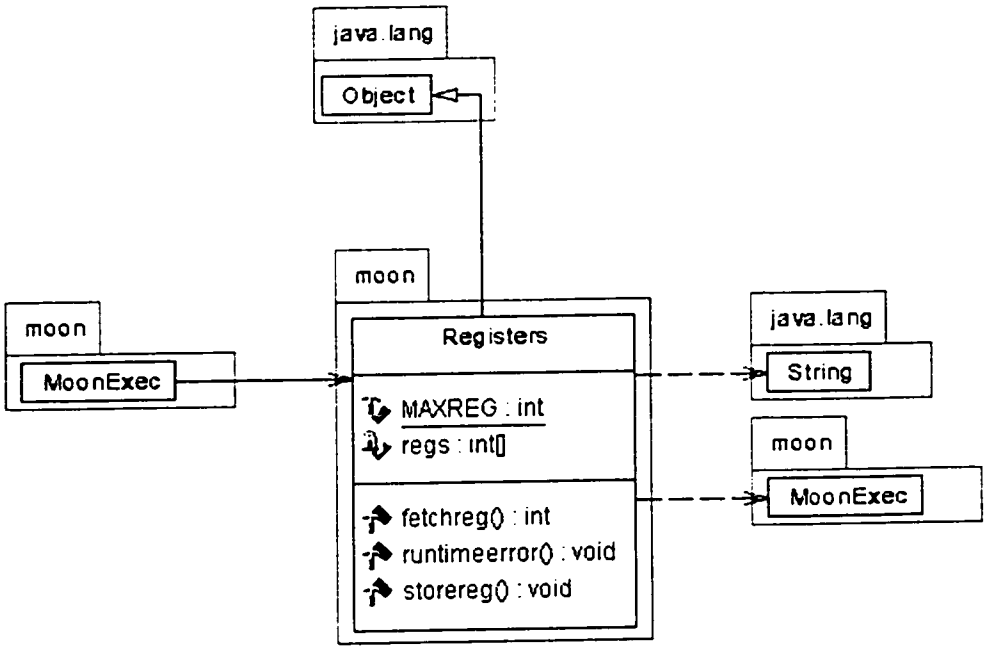


Figure 5.3.6

5.3.7. MoonExec

MoonExec(Figure 5.3.7) is a subclass of **Thread**, which is used for MOON program execution simulation in either normal way or debugging way.

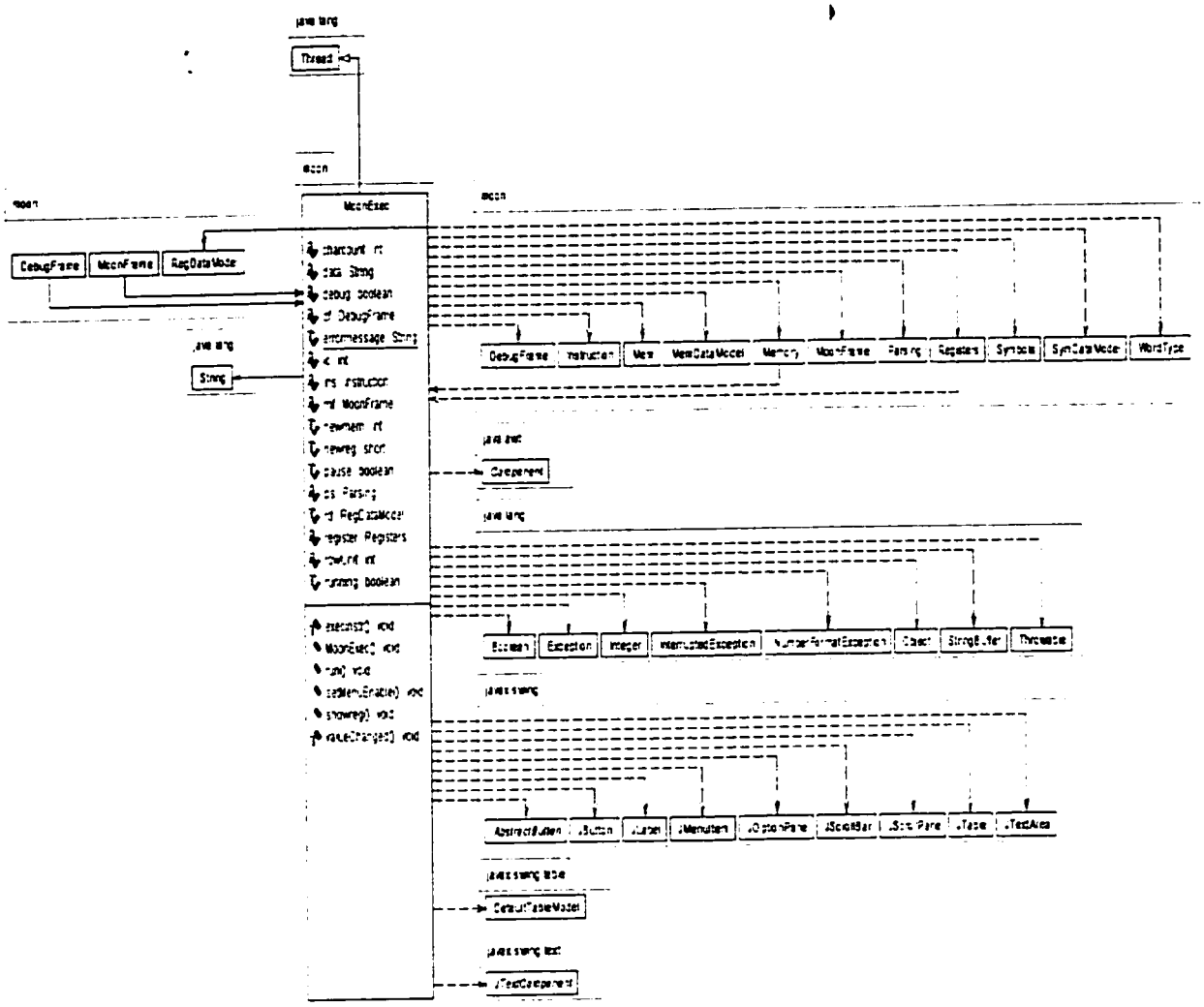


Figure 5.3.7

5.3.7.1. Thread

A Java process is a preemptive lightweight process termed a thread. Java threads make the runtime environment asynchronous, allowing different tasks to be performed concurrently [11]. Implementing threads is achieved in one of two ways: Implementing the **java.lang.Runnable** Interface and Extending the **java.lang.Thread** class. This system uses the last one to implement a **MoonExec** thread. The procedure for extending the **Thread** class is as follows:

1. **MoonExec** extending the **Thread** class overrides the *run()* method from the **Thread** class to define the code executed by the thread.
2. **MoonExec** calls a **Thread** constructor explicitly in its constructors to initialize the thread.
3. The *start()* method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running.

5.3.7.2. Constructor

The **MoonExec** constructor is to:

1. Get the passing argument from **MoonFrame** and set its references representing the displaying tables.
2. Set instruction counter (IC) to the entrypoint and check that it is valid.
3. Check all symbols, and if they contain errors, stop and return.

5.3.7.3. run()

This method contains the code that "does the real work" of a thread.

1. Let thread sleep for 4 milliseconds so that system can have time to handle other things.
2. Executing MOON instructions one by one by invoking method *execinstr()*.
3. If it is running under debugging, it will show registers table.

5.3.7.4. execinstr()

execinstr() is to execute the instruction which is located at the address specified by IC in the simulated memory.

1. If the memory word contains flag that indicates a breakpoint, the *Pause* will be true.

2. If *Pause* is true, it will check whether user has updated values of registers, symbols and memory tables and do corresponding updates.
3. It fetches an instruction and according to formats(A or B) and operands codes to do operations.

5.3.8. MoonFrame

MoonFrame is the class that provides the graphical user interface and many important functions to make the system work. It is the main part of MOON Debugger/Simulator that ensures that: loading and editing file, assembling source code, and running MOON program etc. are included.

5.3.8.1. Event Delegation

GUI applications are event-driven. User interaction with the GUI results in events being generated to inform the application of user actions. Event handling in Java is based on the event delegation model[11]. Handling events in a GUI application, using the event delegation model, can be divided into the following two tasks when building the application:

- Setting up the propagation of events from event sources to event listeners.
- Providing the appropriate actions in event listeners to deal with the events received.

The Figure 5.3.8.1 shows how events(user clicking menu 'new' of **MoonFrame**) are delivered from the source to the listener, then **MoonFrame** sets *jTextAreaEdit* window to blank.

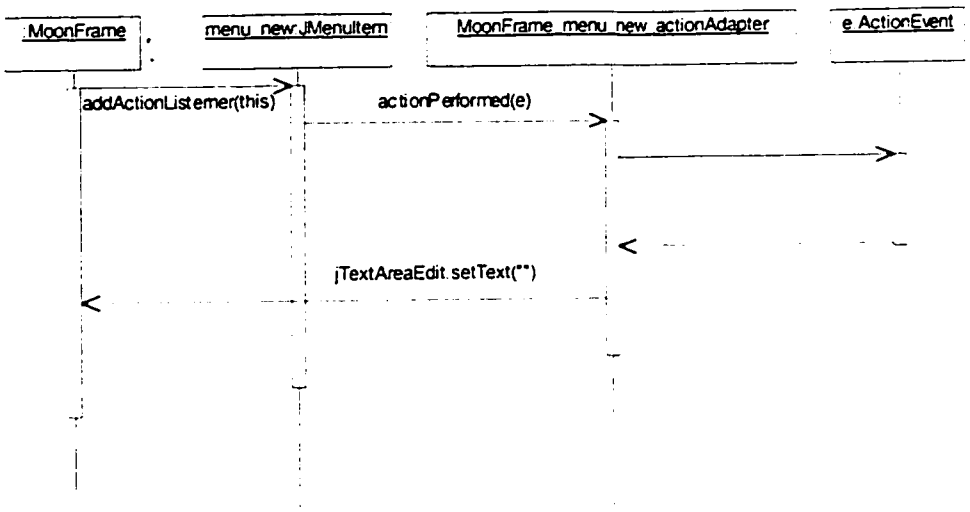


Figure 5.3.8.1

MoonFrame creates 15 inner listener classes for different GUI components event listening. **MoonFrame_menu_new_actionAdapter** mentioned above is one of them.

5.3.8.2. Editor

The MOON source code editing tool is achieved by implementing **Document** from *javax.swing.text* and **JTextArea** from javax.swing.JTextArea. The **Document** is a container for text that serves as the model for swing text components. All documents need to be able to add and remove simple text. Typically, text is inserted and removed via gestures from a keyboard or a mouse. What effect the insertion or removal has upon the document structure is entirely up to the implementation of the document. **JTextArea** has a method called *getDocument* that fetches the model associated with the editor and minimal amount of state required to be a text editor. The diagram of event listener of the editor is shown in Figure 5.3.8.2.

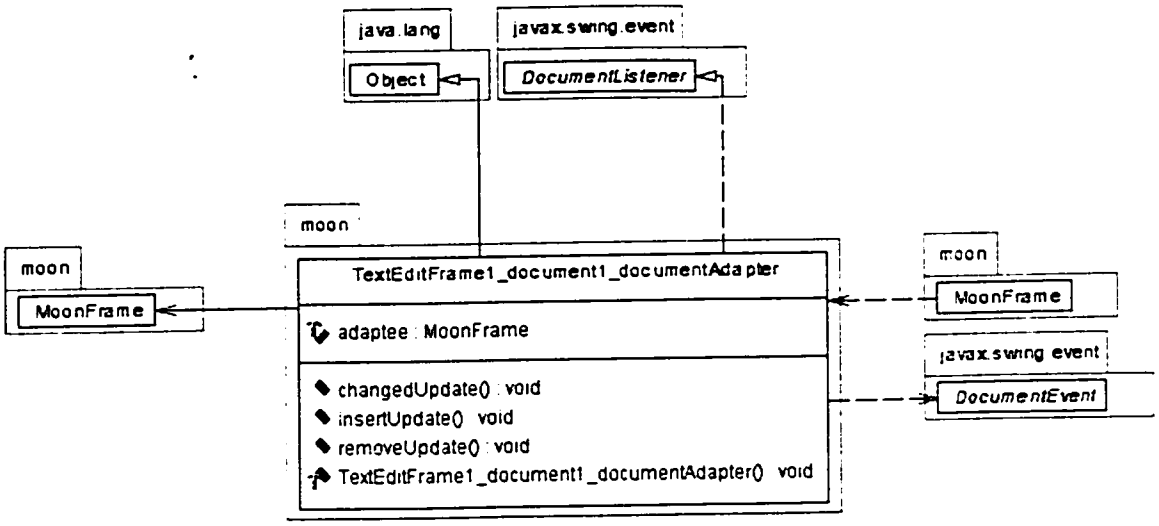


Figure 5.3.8.2

5.3.8.3. File

The menu "File" has submenu items which are "New", "Load", "Save", "Save as" and "Exit".

- "New": set the Editor window content to blank and the flag(*dirty*) value is *false* that means file is not changed up to now.
- "Load": invoke the method of **JFileChooser** *showOpenDialog(this)*, and run defined function *loadFile()* to load a file from the disk. *LoadFile()* will create a character array of the size of the file to use as a data buffer, into which we will read the text data, then the text buffer will be appended to the editor window.
- "Save" and "Save as": Invoke method *saveFile()* and *saveAsFile()* to save the text in editor window to a file. The difference between *saveFile()* and *saveAsFile()* is if the value of current file name-*currFileName* is *null*.

5.3.8.4. Run

Assembling successfully is required before execution of MOON program can begin. For this reason, all the other tasks under "Run" menu should be disabled except "assemble".

In method *menu_assemble_actionPerformed(ActionEvent e)*, the object of **Parsing** is created. If no errors are reported by the **Parsing**, "Run" task can be enabled and system will create a thread of **MoonExec**; In addition, "Debug" is also enabled for system to initiate **DebugFrame**.

5.3.9. Debug Frame

DebugFrame(Figure 5.3.9) is a class for showing a window by which user can run and debug MOON program, and do other utilities.

5.3.9.1. Constructor

The constructor of **DebugFrame** is *DebugFrame(MoonFrame parent)* that passes a **MoonFrame** object and contains a method *jbInit()* to initialize necessary attributes and GUI components:

1. Create registers, memory and symbols table model data and add these data to their tables.
2. Set all the properties of GUI components including our created tables.

5.3.9.2. Table Model

With the **JTable** class, you can display tables of data, optionally allowing the user to edit the data. **JTable** doesn't contain or cache data; it's simply a view of data. Every table gets its data from an object that implements the **TableModel** interface[12]. Generally, the implementation of table model is in a subclass of the **AbstractTableModel** class. Java has provided an implemented table model-**DefaultTableModel** that can be inherited.

This system has three tables: registers table, symbols table and memory table, so three table models should be established. Memory table model is a subclass of

DefaultTableModel; however symbols table model and registers table model are the subclasses of **AbstractTableModel** and they have to implement methods:

getValueAt(int row, int col), setValueAt(Object aValue, int row, int col), getColumnCount(),getRowCount() and *getColumnName(int col).*

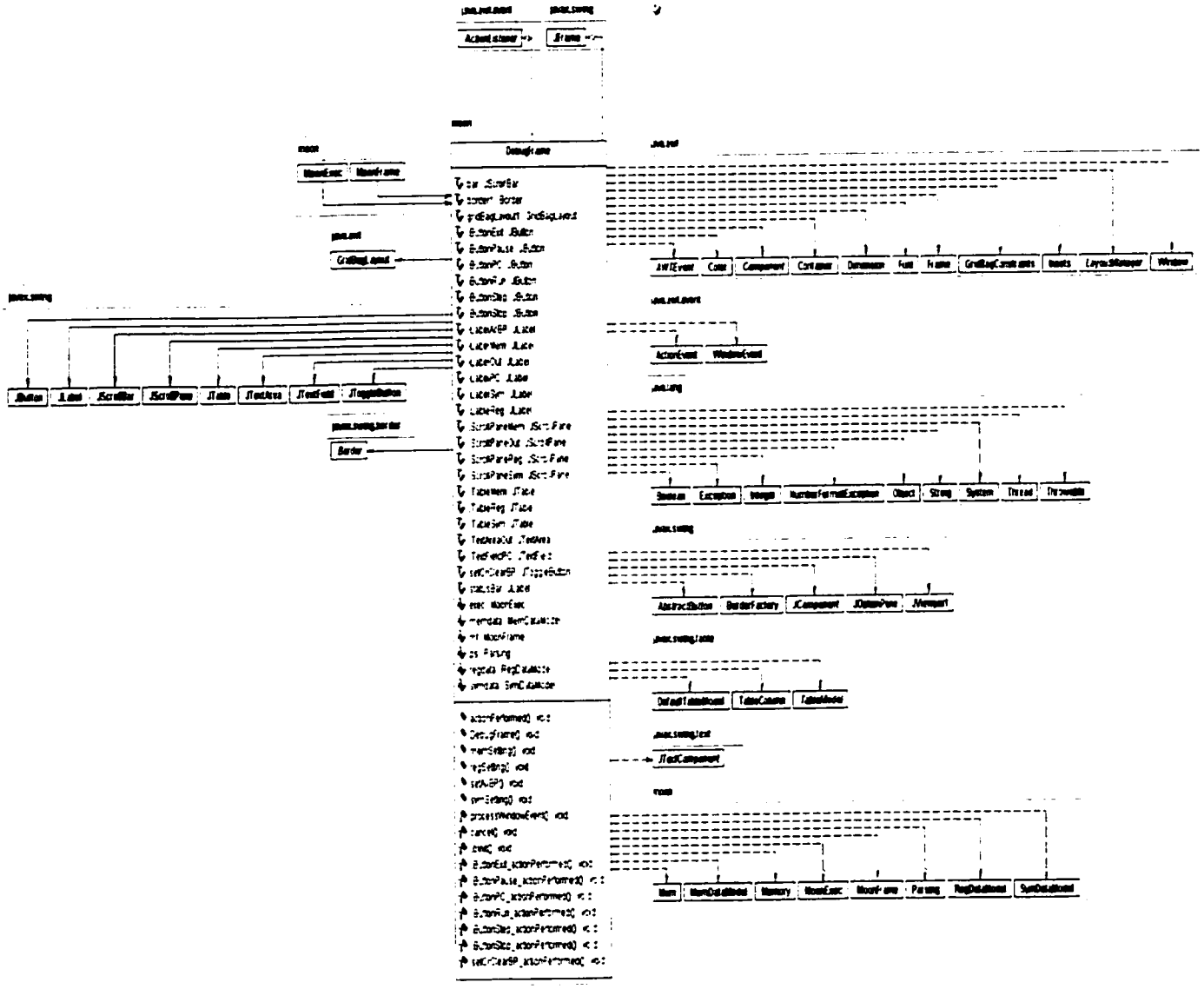


Figure 5.3.9

5.3.9.3. Run, Step, Stop and Pause

The "Run" action creates a thread of **MoonExec**, initializes registers table data and starts of the thread. "Step" action is to set the attribute *pause* of the thread to *false*, so that it will continue to run. In contrast to "Step", "Pause" action sets *pause* to *true*. "Stop" action sets *running* of the thread to *false* in order to let it terminate.

5.3.9.4. PC

DebugFrame provides the function to search memory word by Program Counter(PC) which is also called Instruction Counter(IC) introduced before. It parses what user inputs in the PC text input area to an integer and then lets the scroll bar jump to where the memory word locates and highlight this row in the memory table.

Chapter 6: Conclusion

6.1. Summary

The MOON Debugger/Simulator is a system that can provide a nice friendly user interface and where the user can edit, compile, run MOON programs; furthermore, the user can trace execution in the debugging environment and modify values during execution when the program is stopped at a breakpoint or paused by the user.

This system is written in objected oriented language Java. Java 2 platform provides Java Virtual Machine(JVM) that enable Java program can run on any of OS with it, and Java Application Programming Interface(API) that is a large collection of ready-made software components to provide many useful capabilities, such as GUI widgets.

The MOON architecture contains several components such as registers, memory and instructions. Each of these components is represented as a class in this system. It also provides two GUI frames derived from **JFrame** and other GUI components.

6.2. Future Work

This system extends the old MOON Simulator to the new MOON Debugger/Simulator that resembles a simple and modern debugger. However, there is still much future work to be done.

1. When it loads much MOON programs to memory table and runs in debugger frame, the display may be not correct because of the GUI components limitation

such as **JTable** space limitation (MOON requires up to 4000 rows to be displayed in memory displaying table).

2. Instructions may need to be reviewed and upgraded. Some instructions may cause problem when executing in the system, such as **getc** and **putc**: the effect of running in normal or trace mode is different. In normal mode, input dialogue can get one character that user inputs; unlike normal mode, in trace mode, input dialogue will appear twice: for the first time, user needs input one character, and the second time user clicks 'OK' button without inputting any character for ending the input.
3. More debugging features should be added such as execute the program until condition, and user modify memory content more easily and can do some operations directly.
4. The Help information is not adequate as a guide for this system.

It is expected that most problems are solved and more functions are added in the future. By new object-oriented and Java technology development, this system can become a non error-prone, strong and easy-to-use system.

References

[1] *Peter Grogono, "MOON Document", Computer Science Department, Concordia University, Canada, 1995.*

[2] *Grady Booch, James Rumbaugh, Ivar Jacobson, "The Unified Modeling Language User Guide", Addison-Wesley, 1998.*

[3] *Ian Sommerville, "Software Engineering", Fifth Edition, Addison-Wesley, 1995.*

[4] *Deitel Paul J., "Java How To Program", Third Edition, Prentice Hall, 1999.*

[5] *Steven P. Reiss, "A Practical Introduction to Software Design with C++", John Wiley & Sons, Inc., 1998.*

[6] *John Hunt and Alex McManus, "Key Java-Advanced Tips and Techniques", Springer, 1998.*

[7] *Ahmed Seffah, Course material and slides-"User-Centered Design for Software Engineers:Techniques, Tools and Applications", Computer Science Department, Concordia University, Canada, 2001.*

[8] *A.Seffah and J.Rilling, "Investigation the Relationship between Usability and Conceptual Gaps for Human-Centric CASE Tools", Computer Science Department, Concordia University, Canada, 2001.*

[9] *Nielsen J., and Mack R.L., "Heuristic Evaluation", In Usability Inspection Methods, John Wiley and Sons, 1994.*

[10] *Sun Corporation, "JAVA™ Foundation Classes (JFC)", Website: http://java.sun.com/products/jfc*

[11] *Khalid A. Mughal, Rolf W. Rasmussen, "A Programmer's Guide to Java Certification-A comprehensive primer", Addison-Wesley, 1999.*

[12] *Kathy Waltrth, Mary Campione, "The JFC Swing Tutorial: A Guide to Constructing GUIs", Addison-Wesley, Website:*

http://java.sun.com/docs/books/tutorial/uiswing/index.html

Appendix A: User's Manual

1. Starting the system

In a computer system with JDK1.2 or above installed, find the directory which contains moon subdirectory that have all the MOON Debugger/Simulator Java class files. On this directory, input command: java moon.MoonClass to start the system.

2. Moon Frame Window

It has a menu bar, status bar, text editor window and output window. Menu bar has four menu items: **File, Edit, Run, Help**

2.1. File

File menu has 5 items for the operations on file: **New, Load, Save, Save As, Exit.**

2.1.1. New

Clear the MOON program code and ready to reload file.

2.1.2. Load

Load one file-MOON program to the system or repeat this action to load more files to the system.

2.1.3. Save

Save modified file as current file name

2.1.4. Save As

Save modified file to a new file.

2.1.5. Exit

Close the window and exit from the system

2.2. Edit

Edit menu has 4 items for code editing: **Cut, Copy, Paste, Select All.**

2.2.1. Cut

Select the text with mouse and click this menu items to make the text moved to clipboard.

2.2.2. Copy

Select the text with mouse and click this menu items to make the text copied to clipboard.

2.2.3. Paste

Paste the clipboard content to the specified place in the editor window.

2.2.4. Select All

Make all the text in the editor window to be selected.

2.3. Run

Run menu has 4 items for assembling, debugging and running MOON program: **Assemble, Run, Stop, Debug.**

2.3.1. Assemble

Assemble the code from the editor window. Output window below the editor window will output the compiling error message if the code contains error.

2.3.2. Run

If code is assembled correctly, clicks this item can execute this MOON program and output the result to the output window.

2.3.3. Stop

During the execution, user can stop it immediately.

2.3.4. Debug

Bring a new Debug frame window

3. Debug Frame Window

Debug Frame window has three tables representing memory, symbols, registers, and several buttons, output text window, status bar

3.1. PC

It includes a text input area and an 'OK' button. User can input a number within 0~15999 as the Program Counter(PC) in the input area and click 'OK' to locate and display this memory word in the memory table.

3.2. Run

Click 'Run' button to execute the code, output the result in the output window, and display the variances of registers and memory on their table.

3.3. Exit

Close this Debug Frame window.

3.4. Set/Clear Breakpoints All

Click this button to toggle Set/Clear all the breakpoints for the instructions of this program in the memory table

3.5. Step

Click this button to make the program continue to execute to the next breakpoint.

3.6. Pause

Click this button to make the program execution pause at this point.

3.7. Stop

Click this button to make the program execution stop and PC reset to the 0.

3.8. Tables

There are three tables: memory, symbols and registers.

3.8.1. Memory Table

Memory table is located below the text 'Memory'. It has 5 columns:

Breakpoints - a checkbox that user can check or uncheck to set or clear the breakpoint for this instruction.

PC – it indicates the Program Counter(PC)

Word – it shows the memory word in Text(ASCII).

Word-Hex – it shows the memory word in Hex.

Word-Dec – it shows the memory word in Decimal.

3.8.2. Registers Table

Registers table is located right of the text 'Registers'. It has 4 columns:

No. – it shows the register sequence number.

Dec. – it shows the value of the register in Decimal.

Text - it shows the value of the register in Text(ASCII).

Hex – it shows the value of the register in Hex.

3.8.3. Symbols Table

Symbols table is located below of the text 'Symbols'. It has 3 columns:

Name – it shows the name of the symbol.

Value. – it shows the value of the symbol.

3.8.4. Used At - it shows where the symbols appears.

Appendix B: Source Code

Memory.java

```
/**
 * A word of memory contains an instruction (A or B format), four bytes,
 * or a 32-bit signed integer.  The simulated memory also contains two
 * flags, to tell whether the word contains an instruction and whether
 * it is a breakpoint.
 *
 * The simulated memory is an array of four byte words.  Simulated
 * addresses are byte addresses.  Consequently, all addresses are
 * shifted right 2 bits before being used.  Only the memory module
 * should know about this.  Conventionally,  is a byte address
 * and  is a word address.
 *
 * Each component of the memory array contains:
 *    - word:                    The contents of simulated memory.
 *      cont:          'a' -- format A instruction
 *                                                        'b' -- format B instruction
 *                                                        'd' -- data
 *                                                        'u' -- undefined
 *
 *      breakpoint:     True if this is a breakpoint.
 *
 */

package moon;
import javax.swing.JOptionPane;
public class Memory {
    Memory Size
  final static int MEMSIZE = 1000;
  Mem mem[] = new Mem[MEMSIZE];
  int ic;  *Instruction Counter: contains address of instruction
                  that will be executed next.*
  WordType ir = new WordType();  *Instruction register: contains the instruction
                  that will be executed next. *
  int mar;    *Memory address register. stores word address of last access.*
  WordType mdr = new WordType();  *Memory data register.*

  int entrypoint = -1;  *Address of first instruction *
  String errormessage = new String();
  Instruction ins = new Instruction();
   * Construct(Initialize) the memory: value=zero, kind=undef, no breakpoint.
   * Set "hardware" addresses to an illegal value.
   */
  public Memory() {
     int wordaddr;
     for(wordaddr=0;wordaddr<MEMSIZE;wordaddr++){
        Mem amem = new Mem();
        amem.word.set_data(0);
        amem.cont = 'u';
        amem.breakpoint = false;
        mem[wordaddr] = amem;
        ic = -1;
        mar = -1;
      }
  }
  *Report a run time error if an illegal address is used.*
 boolean outofrange(int addr){
    if (addr<0 ||(addr >>2) > MEMSIZE) {
```

```
    runtimeerror("address error");
    return true;
  }
  return false;
}

/*Report a run time error if a memory word access is not on a
 * four byte boundary. */
boolean misaligned(int addr) {
  if ((addr & 3)!= 0) {
    runtimeerror("alignment error");
    return true;
  }
  return false;
}

/*Fetch the instruction at ic, store it in ir, and increment ic.*/
short fetch() {
  char cont;
  if (outofrange(ic)) {
    ir.set_data(0);
    return 0;
  }
  cont = mem[(ic>>2)].cont;
  if (!(cont=='a' || cont == 'b')) {
    runtimeerror("illegal instruction");
    ir.set_data(0);
    return 0;
  }
  ir.set_data(mem[ic>>2].word.get_data());
  ic += 4;
  return (short)cont;
}

/*Fetch a data word from memory and return it. */
int getmemword(int addr) {
  int wordaddr;
  if (outofrange(addr) || misaligned(addr))
    return 0;
  wordaddr = addr >> 2;
  if(wordaddr != mar) {
    mar = wordaddr;
    mdr.set_data(mem[wordaddr].word.get_data());
  }
  return mdr.get_data();
}

/* Store a word in memory. */
void putmemword (int addr, int data) throws Exception{
  int wordaddr;
  if (outofrange(addr) || misaligned(addr))
    return;
  wordaddr = addr >> 2;
  if (mem[wordaddr].cont == 'a' || mem[wordaddr].cont == 'b') {
    runtimeerror("overwriting instructions");
    return;
  }
  mdr.set_data(data);
  mar = wordaddr;
  mem[mar].word.set_data(mdr.get_data());
  mem[mar].cont = 'd';
  return;
}

/* Fetch a byte from memory. */
```

```
byte getmembyte (int addr) {
int wordaddr = addr >> 2;
short offset = (short)(addr & 3);
if (outofrange(addr)) return 0;
if (wordaddr != mar){
  mar = wordaddr;
}
mdr.set_data(mem[mar].word.get_data());
return (mdr.get_byts())[offset];
}

 * Store a byte in memory. *
void putmembyte (int addr, byte byt) throws Exception {
  int wordaddr = addr >> 2;
  short offset = (short)(addr & 3);
  if (outofrange(addr)) return;
  if (mem[wordaddr].cont == 'a' || mem[wordaddr].cont == 'b') {
    runtimeerror("overwriting instructions");
    return;
  }
  mem[wordaddr].word.set_byts((byte)(byt & 255),offset);
  mem[wordaddr].cont = 'd';
}

 * Store an instruction in memory. Used only by loader. *
void putmeminstr(int addr, WordType word, char cont) throws Exception{
  if ((addr & 3)!=0)
    throw new Exception("alignment error");
  else {
    int wordaddr = addr>>2;
    mem[wordaddr].word.set_data(word.get_data());
    mem[wordaddr].cont = cont;
  }
}

 * Store a character in memory. Used only by loader. *
void putmemchar (int addr, byte byt, char cont) {
  int wordaddr = addr>> 2;
  mem[wordaddr].word.set_byts((byte)byt,addr&3);
  mem[wordaddr].cont = cont;
}

 *Report a run time error and stop the program.*
void runtimeerror(String message) {
  MoonExec.errormessage = "Run-time error:" + message;
}
}
```

Parsing.java

```
*Parsing*

package moon;

import java.util.*;
import javax.swing.JOptionPane;

public class Parsing {
  String message = new String();
  Memory memory = new Memory();
  private Token token= new Token();
  private String oldval;
  String errmes = new String();  *Error message*
  private static int errorcount;  *Number of errors detected. *
```

```
private static int count;
private int bci=0; /*Buffer character index*/
private String buffer;
/*Symbols
Symbols sym = new Symbols(memory);
private int addr = 0;
private int linenum = 0;
private Instruction ins= new Instruction();
/*Define Memory table data
MemDataModel md;
/*Define Symbols talbe data
SymDataModel sd;
public Parsing(String s){
  errorcount = 0;
  /*Tokenized the Moon source code by string
  StringTokenizer tbf = new StringTokenizer(s,"\n\r");
  int i=0;
  while (tbf.hasMoreTokens()) {
     buffer = tbf.nextToken();
     linenum++;
     readline();
     bci = 0;
  }
    Store the symbols to a linklist
  try{
    sym.storesymbols();
  }
    catch (Exception e) {
        message += "Runtime Error:" + e + "\n";
  }
    /*dump the memory and symbols to the tables
  dump();
}
 *Record an error for reporting later *
void syntaxerror(String mssg) {
  errorcount++;
  errmes += "Error at: '" + oldval + " " + token.symval
            + "': " + mssg + "\n" + String.valueOf(linenum)
            + " " + buffer + "\n";
}

 *True if character can occur in a symbol*
boolean issymchar(char c) {
  return (Character.isLetterOrDigit(c) || c=='_');
}

 *True if the string is a valid register name. *
boolean isreg(String p) {
  int regnum =0;
  char c=0;
  if (p.length()<=0) return false;
  if (!(p.charAt(0)=='R' || p.charAt(0)=='r')) return false;
  for (int i=1;i<p.length();i++) {
    if (Character.isDigit(p.charAt(i)))
      regnum = 10*regnum + Character.getNumericValue(p.charAt(i));
    else return false;
  }
  if (regnum < Registers.MAXREG) {
    token.reg = regnum;
    return true;
  }
  else {
    syntaxerror("Illegal symbol");
```

```
    return false;
  }
}

/* Read a token and store appropriate values in the class `token'.
 * For error reporting, the token representing the string must be left
 * in `token.symval'.
 */
void next() {
      char[] ch = new char[100];
         int op;
      oldval = new String(token.symval);
      token.symval = new String();
      for(;bci<buffer.length();bci++) {
        if (buffer.charAt(bci) == ' ' || buffer.charAt(bci)=='\t')
          continue;
        else break;
      }

      token.pos = bci;
     if (bci < buffer.length()) {
        if the first char in the token is letter, go
         if (Character.isLetter(buffer.charAt(bci))) {
           Read a register, op code, directive, or symbol
           for(;bci<buffer.length();bci++) {
          if (issymchar(buffer.charAt(bci)))
            token.symval += buffer.charAt(bci);
          else break;
        }
        if (isreg(token.symval)) {
          token.kind = Tokentype.T_REG;
             return;
          }
          for (op = OpType.lw; op < OpType.last; op++) {
          if (token.symval.equals(Instruction.opnames[op])) {
            token.op = op;
               token.kind = Tokentype.T_OP;
            return;
          }
        }
           token.kind = Tokentype.T_SYM;
           return;
         }
      else if (Character.isDigit(buffer.charAt(bci)) || buffer.charAt(bci)=='-'
             || buffer.charAt(bci)=='+' ){
        * Read a signed decimal integer *
          token.symval += buffer.charAt(bci);
        bci++;
        for(;bci<buffer.length();bci++) {
          if (Character.isDigit(buffer.charAt(bci)))
            token.symval += buffer.charAt(bci);
          else break;
        }
        token.intval = Integer.parseInt(token.symval);
           token.kind = Tokentype.T_NUM;
        return;
         }
         else if (buffer.charAt(bci) == '"') {
         * Read a character string enclosed in quotes *
           bci++;
           for(;bci<buffer.length();bci++) {
             if (buffer.charAt(bci) == '"') {
               token.kind = Tokentype.T_STR;
```

```
            bci++;
                break;
            }
            /*no '"' found*/
              if (bci==buffer.length()) {
              token.kind = Tokentype.T_BAD;
              syntaxerror("unterminated string");
              break;
            }
            token.symval += buffer.charAt(bci);
          }
            return;
          }
          else if (buffer.charAt(bci) == ',') {
          bci++;
          token.kind = Tokentype.T_COMMA;
          token.symval = new String(",");
             return;
        }
          else if (buffer.charAt(bci) == '(') {
          bci++;
          token.kind = Tokentype.T_LP;
             token.symval = new String("(");
          return;
          }
          else if (buffer.charAt(bci) == ')') {
          bci++;
          token.kind = Tokentype.T_RP;
             token.symval = new String(")");
          return;
          }
        else if (buffer.charAt(bci) == '%' || buffer.charAt(bci)=='\r') {
          token.kind = Tokentype.T_NULL;
             token.symval = new String("");
          return;
          }
          else {
            token.kind = Tokentype.T_BAD;
          token.symval = new String("");
        }
      }
      else {
        token.kind = Tokentype.T_NULL;
        token.symval = new String("");
        return;
      }
    }
  /* Match a token */
  void match (int kind) {
    if (token.kind == kind) {
      next();
    return;
    }
    switch (kind) {
      case Tokentype.T_COMMA:
             syntaxerror("',' expected");
          break;
      case Tokentype.T_LP:
          syntaxerror("'(' expected");
          break;
      case Tokentype.T_RP:
          syntaxerror("')' expected");
          break;
```

```
    default:
        syntaxerror("Syntax error");
        break;
    }
  }

 * Parse an opcode.  The error should never occur, since this function
 * is called only when the token type is known.
 *
 int getop() {
   if (token.kind == Tokentype.T_OP) {
     int res = token.op;
     next();
     return res;
   }
   syntaxerror("Opcode expected");
   return 0;
 }

  * Parse a register and return the register number *
 int getreg() {
   if (token.kind == Tokentype.T_REG) {
     int res = token.reg;
     next();
     return res;
   }
   syntaxerror("Register expected");
   return 0;
 }

  * Parse a constant (number or symbol) and return value *
 int getint (int addr) {
   if (token.kind == Tokentype.T_NUM) {
     int res = token.intval;
     next();
     return res;
   }
   else if (token.kind == Tokentype.T_SYM) {
     svm.usesymbol(token.symval, addr);
     next();
     return 0;
   }
   syntaxerror("Constant expected");
   return 0;
   }

  * Similar to getint(), but checks that its value can be stored
  * in 16 bits.
  *
 short getshort(int addr) {
   short val = (short)getint(addr);
   if (Math.abs(val) <= 32767)
     return val;
   syntaxerror("Value cannot be represented with 16 bits");
   return 0;
 }

  * Parse one line of source code from the buffer. *
 void readline () {
   int c;
   WordType word = new WordType();
   word.set_data(0);
   next();
```

```
while (token.kind == Tokentype.T_SYM) {
  sym.defsymbol(token.symval, addr);
  next();
}
try {
  if (token.kind == Tokentype.T_OP) {
    switch (token.op) {
     * Format A    registers only *
        * Operands Ri, Rj, Rk *
        case OpType.add:
        case OpType.sub:
        case OpType.mul:
        case OpType.div:
        case OpType.mod:
        case OpType.and:
        case OpType.or:
        case OpType.ceq:
        case OpType.cne:
        case OpType.clt:
        case OpType.cle:
        case OpType.cgt:
        case OpType.cge:
                word.set_fmta_op(getop());
                    word.set_fmta_ri(getreg());
                   match(Tokentype.T_COMMA);
                   word.set_fmta_rj(getreg());
                    match(Tokentype.T_COMMA);
                    word.set_fmta_rk(getreg());
                    memory.putmeminstr(addr, word, 'a');
                    addr += 4;
                       break;
          * Operands Ri, Rj *
      case OpType.not:
      case OpType.jlr:
                       word.set_fmta_op(getop());
                              word.set_fmta_ri(getreg());
                    match(Tokentype.T_COMMA);
                              word.set_fmta_rj(getreg());
                    memory.putmeminstr(addr, word, 'a');
                    addr += 4;
                              break;

       * No operands *
          case OpType.nop:
          case OpType.hlt:
                       word.set_fmta_op(getop());
                              memory.putmeminstr(addr, word, 'a');
                    addr += 4;
                              break;
       * Format B   operands and constant fields *

       * Operands Ri, K(Rj) *
          case OpType.lw:
          case OpType.lb:
                    word.set_fmtb_op(getop());
                              word.set_fmtb_ri(getreg());
                    match(Tokentype.T_COMMA);
                              word.set_fmtb_k(getshort(addr));
                    match(Tokentype.T_LP);
                              word.set_fmtb_rj(getreg());
                              match(Tokentype.T_RP);
                              memory.putmeminstr(addr, word, 'b');
                              addr += 4;
```

```
                    break;

    * Operands K(Rj), Ri *
    case OpType.sw:
    case OpType.sb:
            word.set_fmtb_op(getop());
                    word.set_fmtb_k(getshort(addr));
                    match(Tokentype.T_LP);
                    word.set_fmtb_rj(getreg());
                    match(Tokentype.T_RP);
                    match(Tokentype.T_COMMA);
                    word.set_fmtb_ri(getreg());
            memory.putmeminstr(addr, word, 'b');
                    addr += 1;
                    break;

  * Operands Ri, Rj, K *
    case OpType.addi:
    case OpType.subi:
    case OpType.muli:
    case OpType.divi:
    case OpType.modi:
    case OpType.andi:
    case OpType.ori:
    case OpType.ceqi:
    case OpType.cnei:
    case OpType.clti:
    case OpType.clei:
    case OpType.cgti:
    case OpType.cgei:
                word.set_fmtb_op(getop());
            word.set_fmtb_ri(getreg());
                    match(Tokentype.T_COMMA);
            word.set_fmtb_rj(getreg());
            match(Tokentype.T_COMMA);
                    word.set_fmtb_k(getshort(addr));
            memory.putmeminstr(addr, word, 'b');
            addr += 1;
                    break;

  * Operands Ri, K *
    case OpType.sl:
    case OpType.sr:
    case OpType.bz:
    case OpType.bnz:
    case OpType.jl:
                    word.set_fmtb_op(getop());
            word.set_fmtb_ri(getreg());
            match(Tokentype.T_COMMA);
                    word.set_fmtb_k(getshort(addr));
            memory.putmeminstr(addr, word, 'b');
            addr += 1;
                    break;

  * Operands Ri; *
    case OpType.gtc:
    case OpType.ptc:
    case OpType.jr:
            word.set_fmtb_op(getop());
                    word.set_fmtb_ri(getreg());
             memory.putmeminstr(addr, word, 'b');
                    addr += 1;
                    break;
```

```
 * Operands k *
case OpType.j:
                word.set_fmtb_op(getop());
                      word.set_fmtb_k(getshort(addr));
                      memory.putmeminstr(addr, word, 'b');
                      addr += 4;
                      break;

 * Set the entry point of the program. *
   case OpType.entry:
                    next();
                          if (memory.entrypoint < 0) {
                            memory.entrypoint = addr;
                      break;
                    }
                    syntaxerror("More than one entry point");
                    break;
    * Adjust the address to the next word boundary. *
case OpType.align:
                     next();
                          if ((addr & 3)!=0)
                            addr = (addr & ~3) + 4;
                    break;

* Set the address to the given value. *
    case OpType.org:
                     next();
                          addr = getint(addr);
                          break;

* Store words. *
    case OpType.dw:
                         next();
                         while (token.kind == Tokentype.T_NUM || token.kind == Tokentype.T_SYM) {
                           word.set_data(getint(addr));
                memory.putmeminstr(addr, word, 'd');
                addr += 4;
                          if (token.kind == Tokentype.T_COMMA)
                            next();
                          else
                            break;
                         }
              break;

 * Store bytes *
    case OpType.db:
                         next();
                         while (true) {
              if (token.kind == Tokentype.T_NUM) {
                          if ((0<= token.intval) && (token.intval <= 255)) {
                   memory.putmemchar(addr, (byte)token.intval, 'd');
                   addr++;
                  }
                          else
                            syntaxerror("Value cannot be represented with 8 bits");
                  next();
                         }
                         else if (token.kind == Tokentype.T_STR) {
                  String t = new String(token.symval);
                  byte[] byt = t.getBytes();
                  for(int i=0;i<byt.length;i++) {
                    memory.putmemchar(addr,byt[i], 'd');
```

```
                                addr++;
                }
                next();
                        }
                        if (token.kind == Tokentype.T_COMMA)
                          next();
                        else if (token.kind == Tokentype.T_NULL)
                          break;
                        else {
                        syntaxerror("Syntax error in byte list");
                        break;
                      }
                    }
                              break;
        * Reserve the given number of words. *
          case OpType.res:
                        next();
                              addr += getint(addr);
                              break;

          /* Should never get here. */
        default:
                    syntaxerror("Unrecognized statement");
              break;
        }
      }
    }catch (Exception e) {
      message += "Runtime Error:" + String.valueOf(memory.ic) +" "+ e.getMessage();
      }
    if (token.kind != Tokentype.T_NULL && errorcount < 5) {
       message += "Warning: junk following " + "'" + token.symval + "'\n";
          message += String.valueOf(linenum) +" " + buffer + "\n";
       errorcount++;
    }
}


void dump() {
  String word;
  md = new MemDataModel();
  sd = new SymDataModel();
  if (!errmes.equals("")) return;
  try {
    for (int i = 0;i<(Memory.MEMSIZE<<2);i +=4) {
      word = ins.showword(i,memory);
      Vector row = new Vector(5);
      row.addElement(new Boolean(memory.mem[i>>2].breakpoint));
      row.addElement(Integer.toString(i));
      row.addElement(word);
      row.addElement(Integer.toHexString(memory.mem[i>>2].word.get_data()));
      row.addElement(Integer.toString(memory.mem[i>>2].word.get_data()));
      md.addRow(row);
      row = null;
    }
  }
    catch (Exception e) {
        message += "Runtime Error:" + e + "\n";
        return;
  }
  //Display all symbols and their uses.
  ListIterator sl = sym.symbols.listIterator();
  Symnode p;
  int i = 0;
```

```
    while (sl.hasNext()) {
        p = (Symnode)sl.next();
        sd.setValueAt(p.name, i, 0);
        sd.setValueAt(Integer.toString(p.val), i, 1);
        ListIterator ul = p.uses.listIterator();
        String s = new String();
        while(ul.hasNext())
          s += String.valueOf(ul.next()) +",";
        sd.setValueAt(s, i, 2);
        i++;
    }
  }

  void memupdate() {
    String word;
    if (!errmes.equals("")) return;
    try {
      for (int i = 0;i<Memory.MEMSIZE>>2;i +=1) {
        word = ins.showword(i,memory);
        md.setValueAt(word, i>>2, 2);
        md.setValueAt(Integer.toHexString(memory.mem[i>>2].word.get_data()), i>>2, 3);
        md.setValueAt(Integer.toString(memory.mem[i>>2].word.get_data()), i>>2, 4);
        word = null;
      }
    }
      catch (Exception e) {
         message += "Runtime Error:" + e + "\n";
         return;
    }
  }
}
```

MoonExec.java

```
 * Execution *
package moon;

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.JOptionPane;
import java.awt.*;
public class MoonExec extends Thread{

  short newreg;    *Address of a register that has changed *
  int newmem;      *Address of a memory location that has changed*
  boolean running;  *True is the processor is running, false after errors *
  private Registers register = new Registers();  * Total registers(16)*
  RegDataModel rd = new RegDataModel();
    If error occurs, throw an Exception and output the errormessage
  static String errormessage = new String();
  private Instruction ins = new Instruction();
  private boolean debug;
  private Parsing ps;
  private MoonFrame mf;
  private DebugFrame df;
  boolean pause;
  private int  ic;  Instruction Counter
  private int rowUnit;   scroll bar increment unit

    count the input data byte by byte.
  private int charcount;
    gets input data string
  private String data;
```

```
  Constructor
public MoonExec(MoonFrame mf, boolean debug) {
  running = true;
  pause = false;
  this.debug = debug;
  this.mf = mf;
  this.df = mf.df;
  this.ps = mf.ps;
    Set ic to entrypoint
  ps.memory.ic = ps.memory.entrypoint;
    Get the scroll bar increment value.
  if (df!=null)
    rowUnit = df.bar.getUnitIncrement(1);
    If there is no entrypoint for the Moon code, stop.
  if (ps.memory.ic<0) {
    JOptionPane.showMessageDialog(null,"There is no 'entry' directive.\n",
    "Error ",JOptionPane.ERROR_MESSAGE);
    running = false.
    setMenuEnable(false);
  }
    Check all the symbols, if there is errors, stop.
  String symerr = ps.sym.checksymbols();
  if(!symerr.equals("")) {
    JOptionPane.showMessageDialog(null,symerr + "\n",
    "Error ",JOptionPane.ERROR_MESSAGE);
    running = false;
    setMenuEnable(false);
  }
}

public void run() {
    Running initialize
  mf.jTextAreaROut.setText("");
  setMenuEnable(true);
  mf.statusBar.setText("Running...");
  if (debug) {
      Before running the code, make sure if user has setup breakpoint or
      changed the values of registers and symbols.
    valueChanged();
    df.jTextAreaOut.setText("");
    Execute the instructions. If running false, stop.
  }
  while(running) {
      Let this thread sleep for a while, so that user can do something.
    try {
      Thread.sleep(1);
    } catch (InterruptedException e) {}
    setMenuEnable(true);
    try {
      execinstr();
    }
    catch (Exception e) {
      running = false;
      JOptionPane.showMessageDialog(null,String.valueOf(ps.memory.ic) +" "+ e.getMessage(),
        "Runtime Error ",JOptionPane.ERROR_MESSAGE);
      setMenuEnable(false);
    }
      Display the change of the registers data
    if (debug) {
      if (newreg>=0)
        showreg(newreg);
    }
  }
```

```
    setMenuEnable(false);
    mf.statusBar.setText("Stopped");
  }
  // Execute the instruction at address ic2.
  void execinstr() throws Exception {
    int addr, w1, w2, k, rk, c;
    ic = ps.memory.ic;
    // Check the breakpoint, if it is true, then program will pause here.
    if (ps.memory.mem[ic>>2].breakpoint)
      pause = true;
    // Highlight the executing instruction in the memory table.
    if (debug) {
      df.bar.setValue(rowUnit*((ic>>2) - 5));
      df.jTableMem.setRowSelectionInterval(ic>>2, ic>>2);
    }
    // When the program pauses, user can change the register and symbol values.
    while (pause) {
      valueChanged();
      if (!running) break;
    }
    // Move next instruction to 'ir'.
    short cont = ps.memory.fetch();
    byte ch;
    // If execution finds error, throw an exception, then stops.
    if (!MoonExec.errormessage.equals("")) {
      throw new Exception(MoonExec.errormessage);
    }
    newreg = -1;
    newmem = -1;
    switch((char)cont) {
      /*Format A instructions with register operands. */
      case 'a':
                switch ((char)ps.memory.ir.get_fmta_op()) {
                  /* add Ri, Rj, Rk*/
                  case OpType.add:
                                  register.storereg((short)ps.memory.ir.get_fmta_ri(),
                                    register.fetchreg((short)ps.memory.ir.get_fmta_rj()) +
                                    register.fetchreg((short)ps.memory.ir.get_fmta_rk()));
                                  newreg = (short)ps.memory.ir.get_fmta_ri();
                                  break;
                 /*sub Ri, Rj, Rk */
                  case OpType.sub:
                                  register.storereg((short)ps.memory.ir.get_fmta_ri(),
                                    register.fetchreg((short)ps.memory.ir.get_fmta_rj()) -
                                    register.fetchreg((short)ps.memory.ir.get_fmta_rk()));
                                  newreg = (short)ps.memory.ir.get_fmta_ri();
                                  break;
                 /*mul Ri, Rj, Rk */
                  case OpType.mul:
                                  register.storereg((short)ps.memory.ir.get_fmta_ri(),
                                    register.fetchreg((short)ps.memory.ir.get_fmta_rj()) *
                                    register.fetchreg((short)ps.memory.ir.get_fmta_rk()));
                                  newreg = (short)ps.memory.ir.get_fmta_ri();
                                  break;
                /*div Ri, Rj, Rk */
                  case OpType.div:
                                  rk = register.fetchreg((short)ps.memory.ir.get_fmta_rk());
                                  if (rk==0) throw new Exception("error: division by zero");
                                  register.storereg((short)ps.memory.ir.get_fmta_ri(),
                                    register.fetchreg((short)ps.memory.ir.get_fmta_rj()) / rk);
                                  newreg = (short)ps.memory.ir.get_fmta_ri();
                                  break;
               /*mod Ri, Rj, Rk */
```

```
    case OpType.mod:
                    rk = register.fetchreg((short)ps.memory.ir.get_fmta_rk());
                    if (rk==0) throw new Exception("error: modulus with zero operand");
                    register.storereg((short)ps.memory.ir.get_fmta_ri(),
                      register.fetchreg((short)ps.memory.ir.get_fmta_rj()) % rk);
                    newreg = (short)ps.memory.ir.get_fmta_ri();
                    break;
  /*and Ri, Rj, Rk(32 bit logical AND) */
    case OpType.and:
                    register.storereg((short)ps.memory.ir.get_fmta_ri(),
                      register.fetchreg((short)ps.memory.ir.get_fmta_rj()) &
                      register.fetchreg((short)ps.memory.ir.get_fmta_rk()));
                    newreg = (short)ps.memory.ir.get_fmta_ri();
                    break;
  /*or Ri, Rj, Rk(32 bit logical OR) */
    case OpType.or:
                    register.storereg((short)ps.memory.ir.get_fmta_ri(),
                      register.fetchreg((short)ps.memory.ir.get_fmta_rj())
                      register.fetchreg((short)ps.memory.ir.get_fmta_rk())),
                    newreg = (short)ps.memory.ir.get_fmta_ri();
                    break;
  /*ceq Ri, Rj, Rk(Rj=Rk) */
    case OpType.ceq:
                    if ( register.fetchreg((short)ps.memory.ir.get_fmta_rj())
                         == register.fetchreg((short)ps.memory.ir.get_fmta_rk()))
                         c = 1;
                    else c = 0;
                    register.storereg((short)ps.memory.ir.get_fmta_ri(),c);
                    newreg = (short)ps.memory.ir.get_fmta_ri();
                    break;
  /*cne Ri, Rj, Rk(Rj!=Rk) */
    case OpType.cne:
                    if ( register.fetchreg((short)ps.memory.ir.get_fmta_rj())
                         != register.fetchreg((short)ps.memory.ir.get_fmta_rk()))
                         c = 1;
                    else c = 0;
                    register.storereg((short)ps.memory.ir.get_fmta_ri(),c);
                    newreg = (short)ps.memory.ir.get_fmta_ri();
                    break;
  /*clt Ri, Rj, Rk */
   case OpType.clt:
                    if ( register.fetchreg((short)ps.memory.ir.get_fmta_rj())
                         < register.fetchreg((short)ps.memory.ir.get_fmta_rk()))
                      c = 1;
                    else c = 0;
                    register.storereg((short)ps.memory.ir.get_fmta_ri(),c);
                    newreg = (short)ps.memory.ir.get_fmta_ri();
                    break;
  /*cle Ri, Rj, Rk */
   case OpType.cle:
                    if ( register.fetchreg((short)ps.memory.ir.get_fmta_rj())
                         <= register.fetchreg((short)ps.memory.ir.get_fmta_rk()))
                         c = 1;
                    else c = 0;
                    register.storereg((short)ps.memory.ir.get_fmta_ri(),c);
                    newreg = (short)ps.memory.ir.get_fmta_ri();
                    break;
  /*cgt Ri, Rj, Rk */
   case OpType.cgt:
                    if ( register.fetchreg((short)ps.memory.ir.get_fmta_rj())
                         > register.fetchreg((short)ps.memory.ir.get_fmta_rk()))
                      c = 1;
                    else c = 0;
```

```
                        register.storereg((short)ps.memory.ir.get_fmta_ri(),c);
                        newreg = (short)ps.memory.ir.get_fmta_ri();
                        break;
                *cge Ri, Rj, Rk *
                case OpType.cge:
                        if ( register.fetchreg((short)ps.memory.ir.get_fmta_rj())
                             >= register.fetchreg((short)ps.memory.ir.get_fmta_rk()))
                          c = 1;
                        else c = 0;
                        register.storereg((short)ps.memory.ir.get_fmta_ri(),c);
                        newreg = (short)ps.memory.ir.get_fmta_ri();
                        break;
                *not Ri, Rj(32 bit complement)*
                case OpType.not:
                        if ( register.fetchreg((short)ps.memory.ir.get_fmta_rj())==0)
                          register.storereg((short)ps.memory.ir.get_fmta_ri(),1);
                        else
                          register.storereg((short)ps.memory.ir.get_fmta_ri(),0);
                        newreg = (short)ps.memory.ir.get_fmta_ri();
                        break;
                *jlr Ri, Rj (Jump to register and link)*
                case OpType.jlr:
                        register.storereg((short)ps.memory.ir.get_fmta_ri(),ps.memory.ic);
                        ps.memory.ic = register.fetchreg((short)ps.memory.ir.get_fmta_rj());
                        newreg = (short)ps.memory.ir.get_fmta_ri();
                        break;
                *nop*
                case OpType.nop:
                        break;
                *hlt*
                case OpType.hlt:
                        running = false;
                        if (debug)
                          df.statusBar.setText("Finished");
                        else
                          mf.statusBar.setText("Finished");
                        break;
                }
              break;

        *Format B instructions have a 16 bit immediate operand.*
      case 'b':
                        ...
                        ...
                        ...
            break;
        }
      }
    /Show all the registers
    public void showreg(short regnum) {
      char charbuf[] = new char[4];
      WordType word = new WordType();
      word.set_data(register.fetchreg(regnum));
      rd.setValueAt(String.valueOf(word.get_data()),regnum,1);
      rd.setValueAt(ins.wordtochars(word),regnum,2);
      rd.setValueAt(Integer.toHexString(word.get_data()),regnum,3);
      df.jTableReg.repaint();
    }

    void valueChanged() {
      String regValue,symValue;
      int intRegValue;
        Get the changed values from the register table, and update correspondant registers.
```

```
for (int i=0;i<Registers.MAXREG;i++) {
  try {
    regValue = (String)rd.getValueAt(i,1);
    if (regValue == null||regValue.equals(""))
      regValue = "0";
    intRegValue = Integer.parseInt(regValue);
    register.storereg((short)i,intRegValue);
  }catch (NumberFormatException e) {
    JOptionPane.showMessageDialog(null,e,
      "Runtime Error ",JOptionPane.ERROR_MESSAGE);
    rd.setValueAt("0",i,1);
  }
}
  get breakpoint information and set it in the memory
for (int i=0;i<Memory.MEMSIZE;i++) {
  if (ps.memory.mem[i].cont!='u') {
    boolean breakpoint = ((Boolean)ps.md.getValueAt(i,0)).booleanValue();
    ps.memory.mem[i].breakpoint = breakpoint;
  }
}
  Update Symbols
int intSymValue;
for (int i=0;i<ps.sd.getRowCount()-1;i++) {
  symValue = (String)(ps.sd.getValueAt(i,1));
  if (symValue == null||symValue.equals(""))
    symValue = "0";
  try {
    intSymValue = Integer.parseInt(symValue);
    ps.svm.storesymbols((String)(ps.sd.getValueAt(i,0)),intSymValue);
  catch(NumberFormatException e) {
     JOptionPane.showMessageDialog(null,e,
      "Runtime Error ",JOptionPane.ERROR_MESSAGE);
    }
   catch (Exception e) {
    JOptionPane.showMessageDialog(null,e,
      "Runtime Error ",JOptionPane.ERROR_MESSAGE);
  }

}
  Update memory
ps.memupdate();
}

  Set menu and its submenu items able or disable to have functions
public void setMenuEnable(boolean start) {
  if (start) {
    if (debug) {
      df.jButtonPause.setEnabled(true);
      df.jButtonRun.setEnabled(false);
      df.jButtonStop.setEnabled(true);
      df.jButtonStep.setEnabled(true);
    }
    else {
      mf.menu_assemble.setEnabled(false);
      mf.menu_debug.setEnabled(false);
      mf.menu_runstart.setEnabled(false);
      mf.menu_runstop.setEnabled(true);
    }
  }
  else {
    if (debug) {
      df.jButtonPause.setEnabled(false);
      df.jButtonRun.setEnabled(true);
```

```
        df.jButtonStop.setEnabled(false);
        df.jButtonStep.setEnabled(false);
      }
      else {
        mf.menu_assemble.setEnabled(true);
        mf.menu_debug.setEnabled(true);
        mf.menu_runstart.setEnabled(true);
        mf.menu_runstop.setEnabled(false);

      }
    }
  }

}
```

MoonFrame.java

Moon program code editing, assembling and running GUI

```
package moon;

import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.border.*;
import javax.swing.text.*;
import javax.swing.event.*;

Moon GUI
public class MoonFrame extends JFrame {
  JPanel contentPane;
  JMenuBar menu_moon = new JMenuBar();
  JMenu menu_file = new JMenu();
  JMenuItem menu_exit = new JMenuItem();
  JMenu menu_help = new JMenu();
  JMenuItem menu_about = new JMenuItem();
  JLabel statusBar = new JLabel();
  JScrollPane jScrollPaneEdit = new JScrollPane();
  JScrollPane jScrollPaneROut = new JScrollPane();
  TitledBorder titledBorder1;
  JMenuItem menu_new = new JMenuItem();
  JMenuItem menu_load = new JMenuItem();
  JMenuItem menu_save = new JMenuItem();
  JMenuItem menu_saveas = new JMenuItem();
  JMenuItem menu_lang = new JMenuItem();
  JMenu menu_edit = new JMenu();
  JMenuItem menu_cut = new JMenuItem();
  JMenuItem menu_copy = new JMenuItem();
  JMenuItem menu_paste = new JMenuItem();
  JMenuItem menu_selall = new JMenuItem();
  JMenu menu_run = new JMenu();
  JMenuItem menu_assemble = new JMenuItem();
  JMenuItem menu_runstart = new JMenuItem();
  JMenuItem menu_guide = new JMenuItem();
  JMenuItem menu_runstop = new JMenuItem();
  Border border1;
  TitledBorder titledBorder2;
  JFileChooser jFileChooser1 = new JFileChooser();
  TitledBorder titledBorder3;
  TitledBorder titledBorder4;
```

```
private Point loc;
String currFileName = null;     Full path with filename. null means new / untitled.
boolean dirty = false;     True means modified text.

JTextArea jTextAreaEdit = new JTextArea();
Document document1;
String text = new String("");
Parsing ps;
MoonExec exec;
JTextArea jTextAreaROut = new JTextArea();
JMenuItem menu_debug = new JMenuItem();
GridBagLayout gridBagLayout1 = new GridBagLayout();

DebugFrame df;
 **Construct the frame*
public MoonFrame() {
  enableEvents(AWTEvent.WINDOW_EVENT_MASK);
  try {
    jbInit();
  }
  catch(Exception e) {
    e.printStackTrace();
  }
}

 **Component initialization*
private void jbInit() throws Exception {
  contentPane = (JPanel) this.getContentPane();
  titledBorder1 = new TitledBorder("");
  border1 = BorderFactory.createMatteBorder(1, 1, 1, 1, Color.white);
  titledBorder2 = new TitledBorder("");
  titledBorder3 = new TitledBorder("");
  titledBorder4 = new TitledBorder("");
  document1 = jTextAreaEdit.getDocument();
  contentPane.setLayout(gridBagLayout1);
  this.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
  this.setFont(new java.awt.Font("Monospaced", 0, 12));
  this.setSize(new Dimension(185, 112));
  this.setTitle("Moon Simulator");
  menu_file.setText("File");
  menu_exit.setText("Exit");
  menu_exit.addActionListener(new MoonFrame_menu_exit_ActionAdapter(this));
  menu_help.setText("Help");
  menu_about.setText("About");
  menu_about.addActionListener(new MoonFrame_menu_about_ActionAdapter(this));
  contentPane.setBackground(SystemColor.menu);
  contentPane.setMinimumSize(new Dimension(600, 800));
  contentPane.setPreferredSize(new Dimension(600, 800));
  contentPane.setToolTipText("");
  menu_new.setText("New");
  menu_new.addActionListener(new MoonFrame_menu_new_actionAdapter(this));
  menu_load.setText("Load");
  menu_load.addActionListener(new MoonFrame_menu_load_actionAdapter(this));
  menu_save.setText("Save");
  menu_save.addActionListener(new MoonFrame_menu_save_actionAdapter(this));
  menu_saveas.setText("Save As");
  menu_saveas.addActionListener(new MoonFrame_menu_saveas_actionAdapter(this));
  menu_lang.setText("Moon Language Reference");
  menu_edit.setText("Edit");
  menu_cut.setText("Cut");
  menu_cut.addActionListener(new MoonFrame_menu_cut_actionAdapter(this));
  menu_copy.setText("Copy");
```

```
menu_copy.addActionListener(new MoonFrame_menu_copy_actionAdapter(this));
menu_paste.setText("Paste");
menu_paste.addActionListener(new MoonFrame_menu_paste_actionAdapter(this));
menu_selall.setText("Select All");
menu_selall.addActionListener(new MoonFrame_menu_selall_actionAdapter(this));
menu_run.setText("Run");

menu_assemble.setActionCommand("Assemble");
menu_assemble.setText("Assemble");
menu_assemble.addActionListener(new MoonFrame_menu_assemble_actionAdapter(this));

menu_runstart.setEnabled(false);
menu_runstart.setActionCommand("menu_runstart");
menu_runstart.setText("Run");
menu_runstart.addActionListener(new MoonFrame_menu_runstart_actionAdapter(this));
menu_guide.setText("System Guide");
menu_runstop.setEnabled(false);
menu_runstop.setActionCommand("menu_runstop");
menu_runstop.setText("Stop");
menu_runstop.addActionListener(new MoonFrame_menu_runstop_actionAdapter(this));
jScrollPaneEdit.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
jScrollPaneEdit.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
document1.addDocumentListener(new MoonFrame_document1_documentAdapter(this));
document1.addDocumentListener(new TextEditFrame1_document1_documentAdapter(this));
statusBar.setBorder(BorderFactory.createLoweredBevelBorder());
statusBar.setText("    ");
jScrollPaneROut.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
jScrollPaneROut.setDoubleBuffered(true);
menu_debug.setEnabled(false);
menu_debug.setText("Debug");
menu_debug.addActionListener(new java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e) {
    menu_debug_actionPerformed(e);
  }
});
jTextAreaROut.setEditable(false);
jTextAreaEdit.setNextFocusableComponent(jScrollPaneROut);
menu_file.add(menu_new);
menu_file.add(menu_load);
menu_file.addSeparator();
menu_file.add(menu_save);
menu_file.add(menu_saveas);
menu_file.add(menu_exit);
menu_help.add(menu_lang);
menu_help.add(menu_guide);
menu_help.add(menu_about);
menu_moon.add(menu_file);
menu_moon.add(menu_edit);
menu_moon.add(menu_run);
menu_moon.add(menu_help);
this.setJMenuBar(menu_moon);
menu_edit.add(menu_cut);
menu_edit.add(menu_copy);
menu_edit.add(menu_paste);
menu_edit.addSeparator();
menu_edit.add(menu_selall);
menu_run.add(menu_assemble);
menu_run.addSeparator();
menu_run.add(menu_runstart);
menu_run.add(menu_runstop);
menu_run.addSeparator();
menu_run.add(menu_debug);
contentPane.add(jScrollPaneEdit, new GridBagConstraints(0, 1, 1, 1, 1.0, 1.0
```

```
            ,GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets(0, 8, 3, 8), 131, 278));
    contentPane.add(jScrollPaneROut,  new GridBagConstraints(0, 2, 1, 1, 1.0, 1.0
            ,GridBagConstraints.CENTER, GridBagConstraints.BOTH, new Insets(3, 8, 0, 8), 419, 195));
    jScrollPaneROut.getViewport().add(jTextAreaROut, null);
    contentPane.add(statusBar, new GridBagConstraints(0, 3, 1, 1, 0.0, 0.0
            ,GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(0, 8, 4, 8), 451, -4));
    jScrollPaneEdit.getViewport().add(jTextAreaEdit, null);
    document1.addDocumentListener(new TextEditFrame1_document1_documentAdapter(this));

  }
  **File | Exit action performed*
  public void menu_exit_actionPerformed(ActionEvent e) {
    if (okToAbandon()) {
      System.exit(0);
    }
  }

  public void menu_about_actionPerformed(ActionEvent e) {
    MoonFrame_AboutBox dlg = new MoonFrame_AboutBox(this);
    Dimension dlgSize = dlg.getPreferredSize();
    Dimension frmSize = getSize();
    Point loc = getLocation();
    dlg.setLocation((frmSize.width - dlgSize.width) / 2 + loc.x, (frmSize.height - dlgSize.height) / 2 +
loc.y);
    dlg.setModal(true);
    dlg.show();
  }

    Open named file, read text from file into jTextArea1, report to statusBar.
void loadFile(String fileName)
{
  try
  {
      Open a file of the given name.
    File file = new File(fileName);

      Get the size of the opened file.
    int size = (int)file.length();

      Set to zero a counter for counting the number of
      characters that have been read from the file.
    int chars_read = 0;
    FileReader in = new FileReader(file);
     Create a character array of the size of the file,
     to use as a data buffer, into which we will read
     the text data.
    char[] data = new char[size];

     Read all available characters into the buffer.
    while(in.ready()) {
      // Increment the count for each character read,
      // and accumulate them in the data buffer.
      chars_read += in.read(data, chars_read, size - chars_read);
    }
    in.close();
    jTextAreaEdit.setText(jTextAreaEdit.getText() + new String(data, 0, chars_read));
    this.currFileName = fileName;
    this.dirty = false;
      Display the name of the opened directory+file in the statusBar.
    statusBar.setText("Loaded "+fileName);
  }
  catch (IOException e)
  {
```

```
    statusBar.setText("Error Loading "+fileName);
  }
}

  **Overridden so we can exit when window is closed*
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
      menu_exit_actionPerformed(null);
    }
  }

    Save current file; handle not yet having a filename; report to statusBar.
void fileLoad(){
    if (!okToAbandon()) {
    return;
  }
    Use the OPEN version of the dialog, test return for Approve/Cancel
  if (JFileChooser.APPROVE_OPTION == jFileChooser1.showOpenDialog(this)) {
    Call openFile to attempt to load the text from file into jTextAreaEdit
  loadFile(jFileChooser1.getSelectedFile().getPath());
  }
  this.repaint();
}

boolean saveFile() {

    Handle the case where we don't have a file name yet.
  if (currFileName == null) {
    return saveAsFile();
  }

  try
  {
    // Open a file of the current name.
    File file = new File (currFileName);
    // Create an output writer that will write to that file.
    // FileWriter handles international characters encoding conversions.
    FileWriter out = new FileWriter(file);
    String text = jTextAreaEdit.getText();
    out.write(text);
    out.close();
    this.dirty = false;
    return true;
  }
  catch (IOException e) {
    statusBar.setText("Error saving "+currFileName);
  }
  return false;
}

  // Save current file, asking user for new destination name.
  // Report to statusBar.
boolean saveAsFile() {
    // Use the SAVE version of the dialog, test return for Approve/Cancel
  if (JFileChooser.APPROVE_OPTION == jFileChooser1.showSaveDialog(this)) {
    // Set the current file name to the user's selection,
    // then do a regular saveFile
    currFileName = jFileChooser1.getSelectedFile().getPath();
    // repaints menu after item is selected
    this.repaint();
    return saveFile();
  }
  else {
```

```
    this.repaint();
    return false;
  }
}

  void menu_new_actionPerformed(ActionEvent e) {
    if (okToAbandon()) {
      // clear the text in the text editor
      jTextAreaEdit.setText("");
      // clear the current filename and set the file as clean:
      currFileName = null;
      dirty = false;
    }
  }
  // Check if file is dirty.
  // If so get user to make a "Save? yes no cancel" decision.
  boolean okToAbandon() {
    if (!dirty) {
      return true;
    }
    int value = JOptionPane.showConfirmDialog(this, "Save changes?",
                                      "Moon", JOptionPane.YES_NO_CANCEL_OPTION);

    switch (value) {
      case JOptionPane.YES_OPTION:
        // yes, please save changes
        return saveFile();
      case JOptionPane.NO_OPTION:
        // no, abandon edits
        // i.e. return true without saving
        return true;
      case JOptionPane.CANCEL_OPTION:
      default:
        // cancel
        return false;
    }
  }


  void menu_load_actionPerformed(ActionEvent e) {
    // Handle the File Open menu item.
    if (!okToAbandon()) {
      return;
    }
    // Use the OPEN version of the dialog, test return for Approve Cancel
    if (JFileChooser.APPROVE_OPTION == jFileChooser1.showOpenDialog(this)) {
      // Display the name of the opened directory+file in the statusBar.
      statusBar.setText("Loaded "+jFileChooser1.getSelectedFile().getPath());
      // Call loadFile to attempt to load the text from file into JTextArea
      loadFile(jFileChooser1.getSelectedFile().getPath());
      // repaints menu after item is selected
      this.repaint();
    }
  }

  void menu_save_actionPerformed(ActionEvent e) {
    saveFile();
  }

  void menu_saveas_actionPerformed(ActionEvent e) {
    saveAsFile();
  }
```

```
  void jButtonLoad_actionPerformed(ActionEvent e) {
     fileLoad();
  }
  void jButtonSave_actionPerformed(ActionEvent e) {
    saveFile();
  }

  void document1_changedUpdate(DocumentEvent e) {
    dirty=true;
  }

  void document1_insertUpdate(DocumentEvent e) {
    dirty=true;
  }

  void document1_removeUpdate(DocumentEvent e) {
    dirty=true;
  }

  void menu_cut_actionPerformed(ActionEvent e) {
    jTextAreaEdit.cut();
  }

  void menu_copy_actionPerformed(ActionEvent e) {
    jTextAreaEdit.copy();
  }

  void menu_paste_actionPerformed(ActionEvent e) {
    jTextAreaEdit.paste();
  }

  void menu_selall_actionPerformed(ActionEvent e) {
    jTextAreaEdit.selectAll();
  }



  void menu_runstart_actionPerformed(ActionEvent e) {
    if (dirty) {
      JOptionPane.showMessageDialog(null, "File has been modified.\nPlease Assemble at first!",
        "Assemble", JOptionPane.ERROR_MESSAGE);
      return;
    }
    exec = new MoonExec(this, false);
    exec.start();
  }

  void menu_assemble_actionPerformed(ActionEvent e) {
    clear();
    // get source code
    text = jTextAreaEdit.getText();
    statusBar.setText("Assembling" );
    // Parse the source file
    ps = new Parsing(text);
    jTextAreaROut.setText(ps.errmes + ps.message);
    if(!ps.errmes.equals("") || !ps.message.equals("") ) {
      JOptionPane.showMessageDialog(null, "Assembling cannot be
completed!", "Assemble", JOptionPane.ERROR_MESSAGE);
      ps = null;
      menu_debug.setEnabled(false);
      menu_runstart.setEnabled(false);
      menu_runstop.setEnabled(false);
    }
```

```
    else {
      menu_debug.setEnabled(true);
      menu_runstart.setEnabled(true);
      menu_runstop.setEnabled(false);
      statusBar.setText("Assemble Succeed");
    }
    dirty = false;
  }

  void menu_debug_actionPerformed(ActionEvent e) {
    if (dirty) {
      JOptionPane.showMessageDialog(null, "File has been modified.\nPlease Assemble at first!",
        "Assemble", JOptionPane.ERROR_MESSAGE);
      return;
    }
    clear();
    loc = this.getLocation();
    df = new DebugFrame(this);
    df.setSize(700, 150);
    df.setLocation(0, loc.y);
    df.show();
    this.setLocation(df.getLocation().x+df.getSize().width, df.getLocation().y);
  }

  void menu_runstop_actionPerformed(ActionEvent e) {
    exec.running = false;
  }

  void clear() {
    if (df!=null) {
      df.dispose();
      setLocation(loc);
      exec = null;
      System.gc();
    }
  }
}

class MoonFrame_menu_exit_ActionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;
  MoonFrame_menu_exit_ActionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_exit_actionPerformed(e);
  }
}


class MoonFrame_menu_about_ActionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;

  MoonFrame_menu_about_ActionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_about_actionPerformed(e);
  }
}


class MoonFrame_menu_new_actionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;
```

```
  MoonFrame_menu_new_actionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_new_actionPerformed(e);
  }
}

class MoonFrame_menu_load_actionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;

  MoonFrame_menu_load_actionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_load_actionPerformed(e);
  }
}

class MoonFrame_menu_save_actionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;

  MoonFrame_menu_save_actionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_save_actionPerformed(e);
  }
}

class MoonFrame_menu_saveas_actionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;

  MoonFrame_menu_saveas_actionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_saveas_actionPerformed(e);
  }
}

class MoonFrame_document1_documentAdapter implements javax.swing.event.DocumentListener {
  MoonFrame adaptee;

  MoonFrame_document1_documentAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void insertUpdate(DocumentEvent e) {
  }
  public void removeUpdate(DocumentEvent e) {
  }
  public void changedUpdate(DocumentEvent e) {
    adaptee.document1_changedUpdate(e);
  }
}

class TextEditFrame1_document1_documentAdapter implements javax.swing.event.DocumentListener {
  MoonFrame adaptee;

  TextEditFrame1_document1_documentAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
```

```
  public void insertUpdate(DocumentEvent e) {
    adaptee.documentl_insertUpdate(e);
  }
  public void removeUpdate(DocumentEvent e) {
    adaptee.documentl_removeUpdate(e);
  }
  public void changedUpdate(DocumentEvent e) {
  }
}

class MoonFrame_menu_cut_actionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;

  MoonFrame_menu_cut_actionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_cut_actionPerformed(e);
  }
}

class MoonFrame_menu_copy_actionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;

  MoonFrame_menu_copy_actionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_copy_actionPerformed(e);
  }
}

class MoonFrame_menu_paste_actionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;

  MoonFrame_menu_paste_actionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_paste_actionPerformed(e);
  }
}

class MoonFrame_menu_selall_actionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;

  MoonFrame_menu_selall_actionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_selall_actionPerformed(e);
  }
}


class MoonFrame_menu_assemble_actionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;

  MoonFrame_menu_assemble_actionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_assemble_actionPerformed(e);
```

```
  }
}

class MoonFrame_menu_runstart_actionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;

  MoonFrame_menu_runstart_actionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_runstart_actionPerformed(e);
  }
}

class MoonFrame_menu_runstop_actionAdapter implements java.awt.event.ActionListener {
  MoonFrame adaptee;

  MoonFrame_menu_runstop_actionAdapter(MoonFrame adaptee) {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
    adaptee.menu_runstop_actionPerformed(e);
  }
}
```

Symbols.java

```
  Create a symbols linklist. Store, get, find, check symbols.
package moon;
import java.util.*;

public class Symbols {
   * The base of the symbol table. *
  LinkedList symbols = new LinkedList();
  private LinkedList useaddr;
  private Symnode p;
  private Memory m;

  public Symbols(Memory m) {
    this.m = m;
    defsymbol("topaddr", 4 * Memory.MEMSIZE);
  }

 * Return a pointer to a symbol entry.  This always succeeds, because
 * it creates a new entry if it can't find a matching entry.
 *
  Symnode findsymbol( String name) {
     ListIterator sl = symbols.listIterator();
     while (sl.hasNext()) {
        p = (Symnode)sl.next();
        // if two names are equal, find it.
        if (p.name.equals(name)) return p;
     }
     // if there is no entry exists, make one
     Symnode pl = new Symnode();
     pl.name = new String(name);
     pl.val = 0;
     pl.defs = 0;
     symbols.add(pl);
     return pl;
  }
```

```
/* Define a symbol. That is, associate the value 'val' with the symbol <name>. */
Symnode defsymbol(String name, int val) {
   p = findsymbol(name);
   p.val = val;
   p.defs++;
   return p;
}

/* Use a symbol. That is, record the fact that the symbol 'name' must *
 *be stored at 'addr'*
void usesymbol(String name, int addr) {
  p = findsymbol(name);
  Integer address = new Integer(addr);
  p.uses.add(address);
}

 * Return the value of a symbol, or -1 if it doesn't exist. *
int getsymbolval(String name) {
  ListIterator sl = symbols.listIterator();
  while (sl.hasNext()) {
      p = (Symnode)sl.next();
        if two names are equal, find it.
      if (p.name.equals(name)) return p.val;
   }
   return -1;
}

*check if the symbols have errors*
String checksymbols() {
  ListIterator sl = symbols.listIterator();
  String s = new String();
  while (sl.hasNext()) {
      p = (Symnode)sl.next();
      if (p.defs == 0) {
        s += "Undefined symbol: " + p.name + "\n";
      }
      else if(p.defs > 1) {
        s +="Redefined symbol: " + p.name + "\n";
      }
  }
  return s;
}

/Store symbols at their respective locations.
void storesymbols() throws Exception{
  ListIterator sl = symbols.listIterator();
   while (sl.hasNext()) {
      p = (Symnode)sl.next();
      ListIterator ul = p.uses.listIterator();
      while(ul.hasNext()) {
        Integer adInt = (Integer)ul.next();
        int addr = adInt.intValue();
        int wordaddr = addr >> 2;
        switch(m.mem[wordaddr].cont) {
          case 'b':
                    m.mem[wordaddr].word.set_fmtb_k((short)p.val);
                    break;
          case 'd':
                    m.mem[wordaddr].word.set_data(p.val);
                    break;
          default:
                    throw new Exception("Symbol storage error!");
```

```
        }
      }
    }
  }

  update a symbole and store symbols at their respective locations.
  void storesymbols(String name, int val) throws Exception{
    ListIterator sl = symbols.listIterator();
    while (sl.hasNext()) {
      p = (Symnode)sl.next();
      if (p.name.equals(name)) p.val= val;
      ListIterator ul = p.uses.listIterator();
      while(ul.hasNext()) {
        Integer adInt = (Integer)ul.next();
        int addr = adInt.intValue();
        int wordaddr = addr >> 2;
        switch(m.mem[wordaddr].cont) {
          case 'b':
                    m.mem[wordaddr].word.set_fmtb_k((short)p.val);
                    break;
          case 'd':
                    m.mem[wordaddr].word.set_data(p.val);
                    break;
          default:
                    throw new Exception("Symbol storage error");
        }

      }
    }
  }
}
```

Instruction.java

```
 * enumerate all the instructions*
package moon;

import java.util.*;

public class Instruction {
  final static String opnames[] =
                    {"", "lw", "lb", "sw", "sb", "add", "sub", "mul", "div", "mod", "and",
                      "or", "not", "ceq", "cne", "clt", "cle", "cgt", "cge", "addi",
                      "subi", "muli", "divi", "modi", "andi", "ori", "ceqi", "cnei",
                      "clti", "clei", "cgti", "cgei", "sl", "sr", "getc", "putc", "bz",
                      "bnz", "j", "jr", "jl", "jlr", "nop", "hlt", "entry", "align",
                      "org", "dw", "db", "res"
                    };
    Show format 'A' instruction word
  private String showfmta(int addr, WordType word) {
    int op = word.get_fmta_op();
    String opcode = opnames[op];
    StringBuffer sb =new StringBuffer();
    switch (op) {
        Operands Rt, Rs, Rk
      case OpType.add:
      case OpType.sub:
      case OpType.mul:
      case OpType.div:
```

```
        case OpType.mod:
        case OpType.and:
        case OpType.or:
        case OpType.ceq:
        case OpType.cne:
        case OpType.clt:
        case OpType.cle:
        case OpType.cgt:
        case OpType.cge:
                        sb.append(alignment(opcode,6));
                        sb.append(' ');
                        sb.append('r');
                        sb.append(word.get_fmta_ri());
                        sb.append(',');
                        sb.append('r');
                        sb.append(word.get_fmta_rj());
                        sb.append(',');
                        sb.append('r');
                        sb.append(word.get_fmta_rk());
                        break;
          Operands Ri, Rj
        case OpType.not:
        case OpType.jlr:
                        sb.append(alignment(opcode,6));
                        sb.append(' ');
                        sb.append('r');
                        sb.append(word.get_fmta_ri());
                        sb.append(',');
                        sb.append('r');
                        sb.append(word.get_fmta_rj());
                        break;
          No operands
        case OpType.nop:
        case OpType.hlt:
                        sb.append(alignment(opcode,6));
                        break;

        return sb.toString();

    Show Format 'B' instruction word
private String showfmtb(int addr, WordType word) {
    int op = word.get_fmtb_op();
    StringBuffer sb = new StringBuffer();
    String opcode = opnames[op];
    switch (op) {
          Operands Ri,k(Rj)
        case OpType.lw:
        case OpType.lb:
                        sb.append(alignment(opcode,6));
                        sb.append(' ');
                        sb.append('r');
                        sb.append(word.get_fmtb_ri());
                        sb.append(',');
                        sb.append(word.get_fmtb_k());
                        sb.append('(');
                        sb.append('r');
                        sb.append(word.get_fmtb_rj());
                        sb.append(')');
                        break;
          Operands k(Rj), Ri
        case OpType.sw:
        case OpType.sb:
                        sb.append(alignment(opcode,6));
```

```
                sb.append(' ');
                sb.append(word.get_fmtb_k());
                sb.append('(');
                sb.append('r');
                sb.append(word.get_fmtb_rj());
                sb.append(')');
                sb.append(',');
                sb.append('r');
                sb.append(word.get_fmtb_ri());
                break;
  Operands Ri,Rj, K
case OpType.addi:
case OpType.subi:
case OpType.muli:
case OpType.divi:
case OpType.modi:
case OpType.andi:
case OpType.ori:
case OpType.ceqi:
case OpType.cnei:
case OpType.clti:
case OpType.clei:
case OpType.cgti:
case OpType.cgei:
                sb.append(alignment(opcode,6));
                sb.append(' ');
                sb.append('r');
                sb.append(word.get_fmtb_ri());
                sb.append(',');
                sb.append('r');
                sb.append(word.get_fmtb_rj());
                sb.append(',');
                sb.append(word.get_fmtb_k());
                break;
  Operands Ri, K
case OpType.sl:
case OpType.sr:
case OpType.bz:
case OpType.bnz:
case OpType.jl:
                sb.append(alignment(opcode,6));
                sb.append(' ');
                sb.append('r');
                sb.append(word.get_fmtb_ri());
                sb.append(',');
                sb.append(word.get_fmtb_k());
                break;
  Operands Ri
case OpType.gtc:
case OpType.ptc:
case OpType.jr:
                sb.append(alignment(opcode,6));
                sb.append(' ');
                sb.append('r');
                sb.append(word.get_fmtb_ri());
                break;
  Operands K
case OpType.j:
                sb.append(alignment(opcode,6));
                sb.append(' ');
                sb.append(word.get_fmtb_k());
                break;
}
```

```
        return sb.toString();
    }
    /*
     * Convert a word to a string of 4 characters.  Non graphics to '.'.
     * Exact output depends on whether the host is big endian or
     * little endian.
     */
    String wordtochars(WordType word) {
        int i;
        byte[] byts = word.get_byts();
        char[] buf = new char[4];
        for (i=0;i<4;i++) {
          char c = (char)byts[i];
          if (32<=c && c<=126)
             buf[i]=c;
          else buf[i] = '.';
        }
        return String.valueOf(buf);
    }

    /* Display one word of memory.
     */
    String showword (int addr, Memory m) throws Exception {
        char charbuf[] =  new char[5];
        int wordaddr = addr >> 2;
        WordType word = m.mem[wordaddr].word;
        if ((addr & 3) != 0)
            throw new Exception("Internal error: bad address!\n");
        switch(m.mem[wordaddr].cont) {
          case 'a':
                    return showfmta(addr,word);

          case 'b':
                    return showfmtb(addr,word);

          case 'd':
                    return wordtochars(word);
          case 'u':
                    return "??";
          default : return "  ";
        }
    }
    /* Make address column output alignment
     */
    private String alignment(String s, int n) {
      for (int i = s.length();i<n;i++)
        s += " ";
      return s;
    }

}
```